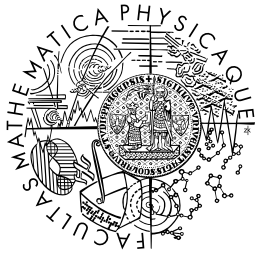

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Aleš Woska

Visual knowledge graph management tool

Department of Software Engineering

Supervisor of the master thesis:	Doc. Mgr. Martin Nečaský, Ph.D.
Study programme:	Software systems
Study branch:	Software engineering

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 28.7.2016 signature of the author

Title: Visual knowledge graph management tool

Author: Bc. Aleš Woska

Department: Department of Software Engineering

Supervisor: Doc. Mgr. Martin Nečaský, Ph.D., Department of Software Engineering

Abstract: Linked data are usually visualized as graph structures which are useful for browsing resources but less useful for viewing structured data. This thesis proposes a solution how to visualize linked data in a tabular structure. The goal of the visualization is to make an orientation in data easier for a user. A tabular structure for specific data can be designed and managed in a graphic editor. Visualized data can be checked for errors and can be edited to obtain a script which fixes them.

Keywords: Linked Data visualization RDF knowledge management

I would like to thank the supervisor of the thesis, Doc. Mgr. Martin Nečaský, PhD. for guidance and support during my work.

Contents

Introduction

Informatics area concerning linked data is a modern way for data operations. Yet, this technology is not very extended. Linked data is a concept used in web technologies which main advantage is a semantic description of data. Most expanded format used for describing Linked data is RDF. RDF format basically describes linked data as triples consists of a subject, a predicate, and an object. RDF triples can be also represented as a graph-like structure. The following example describes data where a city of Prague has population of one million and lies in Czech Republic. Czech Republic is a country with population of 10 million.

Subject	Predicate	Object
Prague	is a	City
Prague	lies in	Czech Republic
Prague	has population	1 000 000
Czech Republic	is a	Country
Czech Republic	has population	1 000 000

Figure 1: Triples representation of linked data in RDF

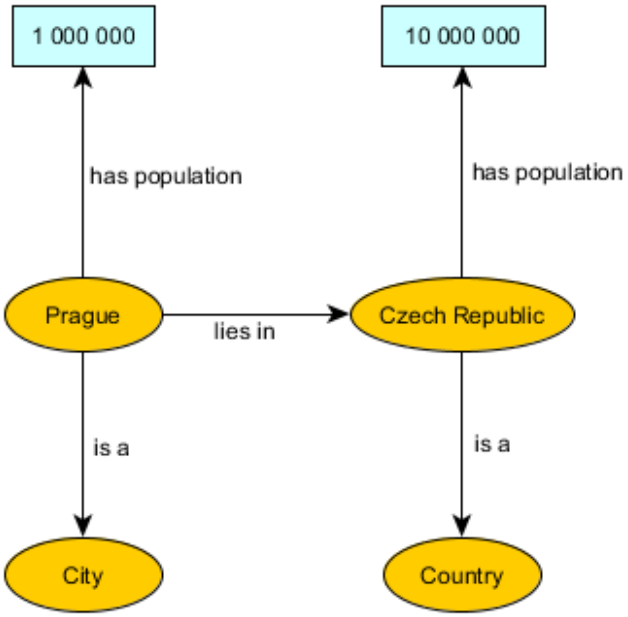

Figure 2: Graph-like representation of linked data in RDF

RDF is only an abstract format. But there exist more specifications and formats around linked data and RDF itself. As RDF is abstract it can be described in many languages, such as JSON-LD, TTL or XML. There exists a schema defining vocabulary for RDF called RDFS. Latest RDFS specification RDFS contains about 30 definitions which can be used to describing RDF. For example prop-

erties like domain, type, subclass, label etc. and classes like literal, container, sequence, class and others.

For queries over RDF data there exists a query language called SPARQL [6]. It's syntax is very similar to SQL but the main difference is that SPARQL is based on describing triples. There are also other query languages for RDF (for example DQL [7], N3QL [8], RDQL [9]) but SPARQL is used the most and it's recommended by W3C as a standard. The query in the example below selects all cities situated in Czech Republic. In WHERE clause we are describing a triples which have the predicate 'rdf:liesIn' and the object 'CzechRepublic'. The result is a list of all such subjects labeled as 'city'. And the result is limited to show only the first 20 entries.

```
SELECT ?city WHERE { ?city rdf:liesIn 'CzechRepublic' . } LIMIT 20
```

Figure 3: Example of SPARQL

There are several ways to visualize RDF data which advantages and disadvantages are explained in the following chapter:

- Table of RDF triples
- Visualizing a query result
- RDF data browser
- Showing through a template
- Showing in a fixed predefined structure.

The thesis is focused on the last way to visualize RDF data - by showing them in a fixed predefined layout.

In the first chapter 1 requirements are summarized, then analyzed and structured into a requirement diagram. There exist other tools working with RDF. Comparison of this thesis with other existing tools and the highlight of its benefits are described in chapter 2. The design of the solution is described in chapter 3. The design includes suggestions of how to satisfy the requirements and a design of the application. Chapter 4 deals with the application testing and defines acceptance criteria for the application. The last chapter 5 contains a user documentation in a form of tutorials of how to work with the application.

Vocabulary

At first we will define a special vocabulary for this thesis, because a lot of used terms are highly overdriven in other areas of computer science. The use of these terms without a proper specification can be quite misleading.

The application will always mean the application created as a part of this thesis. A reference to some general application (not a part of the thesis) will be described as *an application*.

SPARQL endpoint is a service that enables querying via the SPARQL language to a RDF knowledge base.

RDF describes resources. A resource is an *object* identified by an URI. In RDF triple it is the subject. Other parts of a RDF triple is a predicate and an object. If a subject is taken as an object, then a predicate describes it's *property* and an object is its value. This value can be literal and in that case this whole triple describes a *literal property*. But the value can be another URI. That means this triple is describing a relation between two objects and then this triple describes an *object property*.

Type of an *object* is a value defined by the 'rdf:type' property. This property describes resource's class. An *instance* of a class (or by other words, instance of a type) is an object of that type. An instance is visualized as a row in a table.

1. Requirements analysis

1.1 Basic requirements

Browsing data represented in RDF can be difficult and suitable only for a RDF specialist. Other users need some form of superstructure to understand the data. There are several ways to visualize RDF data:

- Table of RDF triples
- Visualizing a query result
- RDF data browser
- Showing through a template
- Showing in a fixed predefined structure

In the following figure there is a more complex example of data represented in RDF. We show advantages and disadvantages of mentioned ways of visualization on this example.

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix ex: <http://www.example.org/> .

ex:Prague rdf:type ex:City .
ex:Prague ex:population '1281152' .
ex:Prague ex:area '496' .
ex:Prague ex:liesIN ex:CzechRepublic .
ex:Prague rdf:type ex:City .
ex:Brno rdf:type ex:City .
ex:Brno ex:population '400933' .
ex:Brno ex:area '230.22' .
ex:Brno ex:liesIN ex:CzechRepublic .
ex:Innsbruck rdf:type ex:City .
ex:Innsbruck ex:population '126851' .
ex:Innsbruck ex:area '104.91' .
ex:Innsbruck ex:liesIn ex:Austria .
ex:Austria rdf:type ex:Country .
ex:Austria ex:name 'Austria' .
ex:Austria ex:population '8623073' .
ex:Austria ex:lang 'German' .
ex:Austria ex:currency ex:Eur .
ex:CzechRepublic rdf:type ex:Country .
ex:CzechRepublic ex:name 'Czech Republic ' .
ex:CzechRepublic ex:population '10538275' .
ex:CzechRepublic ex:lang 'Czech' .
ex:CzechRepublic ex:currency ex:Czk .
ex:Germany rdf:type ex:Country .
ex:Germany ex:name 'Germany' .
ex:Germany ex:population '81292428' .
ex:Germany ex:lang 'German' .
ex:Germany ex:currency ex:Eur .
ex:UnitedKingdom rdf:type ex:Country .
ex:UnitedKingdom ex:name 'United Kingdom' .
ex:UnitedKingdom ex:population '62716000' .
ex:UnitedKingdom ex:lang 'English' .
ex:UnitedKingdom ex:currency ex:Gbp .
ex:Eur rdf:type ex:Currency .
ex:Eur ex:name 'EUR' .
ex:Eur ex:symbol '\EUR{}' .
ex:Czk rdf:type ex:Currency .
ex:Czk ex:name 'CZK' .
ex:Czk ex:symbol 'Kč' .
ex:Gbp rdf:type ex:Currency .
ex:Gbp ex:name 'GBP' .
ex:Gbp ex:symbol '\pounds' .
```

Figure 1.1: More complex RDF example

Table of triples

This method is not very different from a native RDF but at least it is more human readable.

Subject	Predicate	Object
ex:Prague	rdf:type	ex:City
ex:Prague	ex:population	1281152
ex:Prague	ex:area	496
ex:Prague	ex:liesIn	ex:CzechRepublic
ex:Prague	rdf:type	ex:City
ex:Brno	rdf:type	ex:City
ex:Brno	ex:population	400933
ex:Brno	ex:area	230.22
ex:Brno	ex:liesIn	ex:CzechRepublic
ex:Innsbruck	rdf:type	ex:City
ex:Innsbruck	ex:population	126851
ex:Innsbruck	ex:area	104.91
ex:Innsbruck	ex:liesIN	ex:Austria
ex:Austria	rdf:type	ex:Country
ex:Austria	ex:name	Austria
ex:Austria	ex:population	8623073
ex:Austria	ex:lang	German
ex:Austria	ex:currency	ex:Eur
ex:CzechRepublic	rdf:type	ex:Country
ex:CzechRepublic	ex:name	Czech Republic
ex:CzechRepublic	ex:population	10538275
ex:CzechRepublic	ex:lang	Czech
ex:CzechRepublic	ex:currency	ex:Czk
ex:Germany	rdf:type	ex:Country
ex:Germany	ex:name	Germany
ex:Germany	ex:population	81292428
ex:Germany	ex:lang	German
ex:Germany	ex:currency	ex:Eur
ex:UnitedKingdom	rdf:type	ex:Country
ex:UnitedKingdom	ex:name	United Kingdom
ex:UnitedKingdom	ex:population	62716000
ex:UnitedKingdom	ex:lang	English
ex:UnitedKingdom	ex:currency	ex:Gbp
ex:Eur	rdf:type	ex:Currency
ex:Eur	ex:name	EUR
ex:Eur	ex:symbol	€
ex:Czk	rdf:type	ex:Currency
ex:Czk	ex:name	CZK
ex:Czk	ex:symbol	Kč
ex:Gbp	rdf:type	ex:Currency
ex:Gbp	ex:name	GBP
ex:Gbp	ex:symbol	£

Visualizing a query result

Visualizing a query result is useful mainly for transforming pure RDF data to a demanded format and to transform the result into demanded data model. The advantage is that we can transform data into demanded form but the disadvantage is that we have to know the original data structure.

```
\# Query
SELECT ?country_name as 'Country'
  ?city_name as 'City'
  ?currency_name as 'Currency'
WHERE {
  ?country rdf:type ex:Country .
  ?city rdf:type ex:City .
  ?currency rdf:type ex:Currency .
  ?country ex:name ?country_name .
  ?city ex:name ?city_name .
  ?currency ex:name ?currency_name .
  ?city ex:liesIn ?country .
  ?country ex:currency ?currency .
}
```

This example query results into the table:

Country	City	Currency
Prague	Czech Republic	CZK
Brno	Czech Republic	CZK
Innsbruck	Austria	EUR

RDF data browser

RDF data browser visualizes RDF as graph structure. This kind of visualization is more readable but a user still have to understand RDF data.

Showing in a fixed predefined structure

The last method - showing in a fixed predefined structure is a combination of a RDF data broswe and visualization of a query result. Predefined structure have to be created by a user who understans RDF data, but showing the data itself can be done by any user.

The thesis focuses on the last type of a visualization - showing in a fixed predefined structure. Those data will be viewed by a specialist user. Specialist user means that the user is familiar with data meaning, theirs content and structure, but he doesn't have to understand any of RDF technologies or tools themselves. The requirement is to visualize a meaningful representation of RDF data, therefore the user can work with them easily without a special effort. We were told, that a typical user of the application wants to see a result of some data gathering process and wants to check if gathered data are correct. We talked especially about a law nature or a medicine nature of data. Moreover, the content of the data is substantial, but theirs structure is static or changes only slightly. Others

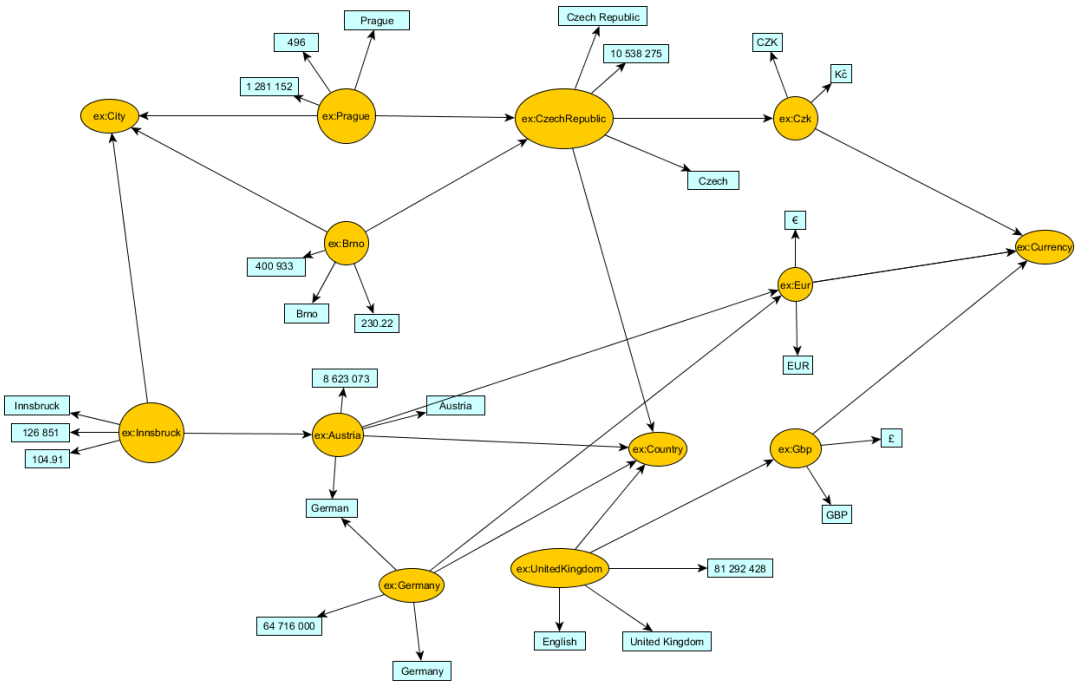

Figure 1.2: RDF data browser example

City

Name	Population	Area
Brno	400 933	230.22
Innsbruck	126 851	104.91
Prague	1 281 152	496

lies in

Country

Name	Language	Population
Austria	German	8 623 073
Czech Republic	Czech	10 538 275
Germany	German	81 292 428
United Kingdom	English	64 716 000

currency

Currency

Name	Symbol
GBP	£
EUR	€
CZK	Kč

Figure 1.3: Showing in a fixed predefined structure example

typical applications show graph-like structure which element's location and context changes as a user moves through the graph. This can be confusing for a user and not very comfortable for a work. It would be useful, if similar objects were shown at the same place and theirs relations and properties can be found on a same location as well. Summarized, same types of objects should be visualized very closed to each other and theirs positions should be fixed. This presumptions leads to a requirement to show objects of the same type in a table where table's row will represent particular instance and table's columns will represent object's properties. Every table will contain only objects of a same type. Tables can be connected by a line. This connection represents an object property between objects of different types.

1.2 Searching and filtering

Table columns headers should contain a text field which will serve as a fulltext filter for a particular property within the table. At first all tables will contain no rows. After entering a few characters the table content should be loaded automatically. This mechanism provides loading data for each table separately, but there must be a way to show connections of objects between tables. After selecting a row in some table in neighbor table only objects connected to the

selected one should be shown. If there is no such objects, there will be loaded for each table. It would be useful to propagate this selection through whole schema and every table. But there could be a problem with data size and for this case a user can choose to propagate the selection through whole schema or just adjacent tables.

1.3 Structure editor

Regarding the position and the appearance of objects in a structure, theirs settings should be editable easily. We mark this tool as the layout editor. There is a need for the layout editor for objects' appearance (like position, size, font or color). Every object may contain many properties. The layout editor have to contain a mechanism to define which properties to show as table columns. For better orientation tables and columns should have some captions. In the layout editor there should be a way to define tables and columns caption. It should be a user defined value or some common property from a RDF ontology. In a similar way there have to be a way to define a connection between tables. This include defining of source and target table and a property that connects them. Individual settings for objects should be able to save for a later use. And as well as for tables a visual properties can be set as well as a caption. There must be a possibility to save current settings for a later use. Saved settings should be listed and be able to use. Previously saved settings should be editable as well. The layout editor can be operated by another user who is more familiar with RDF technologies and therefore can prepare more suitable settings for a specialist user.

1.4 Showing errors

As told in the first paragraph, a user usually wants to check a result of some data gathering process. In such a process there can exists a data that are uncertain and are marked with some kind of an error. If a data structure contains any data that are marked with error, it should be visualized. This applies for both lexicographical errors like a bad string value or a structural error like missing connection between objects or the opposite - there is a connection between objects and it shouldn't. Visualization of these error will serve to a user for fixing them. After data modification a user can mark an error as solved.

1.5 Data editing

Fixing data errors requires their modifications. There are several types of possible operations. First is to edit a simple value (numeric or text). In a context of tables structure it means editing a table cell. Next pair of operations are to add or to remove an object with all its properties which means to add or to remove a table row in a tables structure context. And the last pair of operations are to add or to remove a property between objects of different types. That means to add or to remove a connection between two lines in separate tables (because connection between tables as themselves are defined by the structure). Those operations are considered sufficient to fix all possible errors.

1.6 Changes management

All data modifications should be saved into a temporary memory. Data changes should be able to be reverted in any time. There is no requirement to propagate the changes directly into a data source. A result of data modifications should be a script. If the script is ran on a data source, data should be modified in a same way they were modified in the application. The script will contain of list of individual operations. Only requirement is that the result of the script have to be same as if data were modified directly by the application, but there is no requirement that the script should be optimized in any way.

1.7 Others

There are no requirements for an authorization or an authentication. A user doesn't have to log into the application to work with it. The only authentication needed is to connect to a RDF data source. Nevertheless, the application has to be written in a way to possibly implement authorization and authentication later.

Regarding the application form, at the beginning of the thesis we talked about the thesis as a plugin for the existing project called Payola. Payola is a robust framework developed by MFF UK containing web application with useful tools, such as data sources, data source connectors, data processing tools, filters, visualizers, user control, and other tools for processing RDF data. But during the thesis the Payola framework became outdated and a new project called "Linked Pipes" has begun to be developed. It's concept is markedly different from the Payola and doesn't fit into requirements for the thesis. Because of that we decided to write the application as stand-alone with own frontend, but with possibility to add a connection from the Linked Pipes.

1.8 Use cases and Actors

The analysis discovered two types of roles - a user and a designer. A user only works with data structure itself. He don't work with layouts itself just uses already defined layouts. A designer generalizes a user. In addition he can work with layouts. Beside of that, he can do all operations as a common user do. This descriptions of actors is written in following UML diagram.

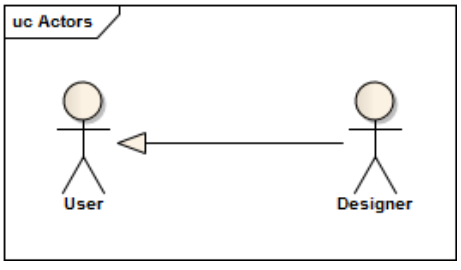

Figure 1.4: Actors

From the analysis there arose some use cases for discovered actors. We've

decided not to use user stories because they have less descripting capabilities. User story is a short description in a common language while use cases provides more technical description and describes communication with a system better. Use case number corresponds to a section number in the current chapter. UC 1 corresponds to the section 1.1, UC 3.* corresponds to the section 1.3, etc.

Use case	Show content of a SPARQL endpoint in a chosen layout.
Number	1
Primary Actor	User
Brief	

Use case	Restrict a view to a selected resource.
Number	2
Primary Actor	User
Brief	Each table column header contains a fulltext filter. This filter is connected to a property in which columns is situated.

Use case	Create a new layout.
Number	3.1
Primary Actor	Designer
Brief	The layout editor is opened. A new layout can be created after entering a name.

Use case	Modify an existing layout.
Number	3.2
Primary Actor	Designer
Brief	There is a list of existing layouts and by clicking layout name the layout editor is opened.

Use case	Delete an existing layout.
Number	3.3
Primary Actor	Designer
Brief	An existing layout can be deleted by clicking a button.

Use case	Show errors in data.
Number	4
Primary Actor	User
Brief	If there exists an error within the data it should be marked.

Use case	Fix data.
Number	5
Primary Actor	User
Brief	Edit a value of a table cell.

Use case	Revert changes.
Number	6.1
Primary Actor	User
Brief	

Use case	Generate an update script.
Number	6.2
Primary Actor	User
Brief	All changes done in a data structures will be processed to generate an update script which performs identical operations.

1.9 Requirements summary

All uses cases are summarized in the following figure using UML use case diagram.

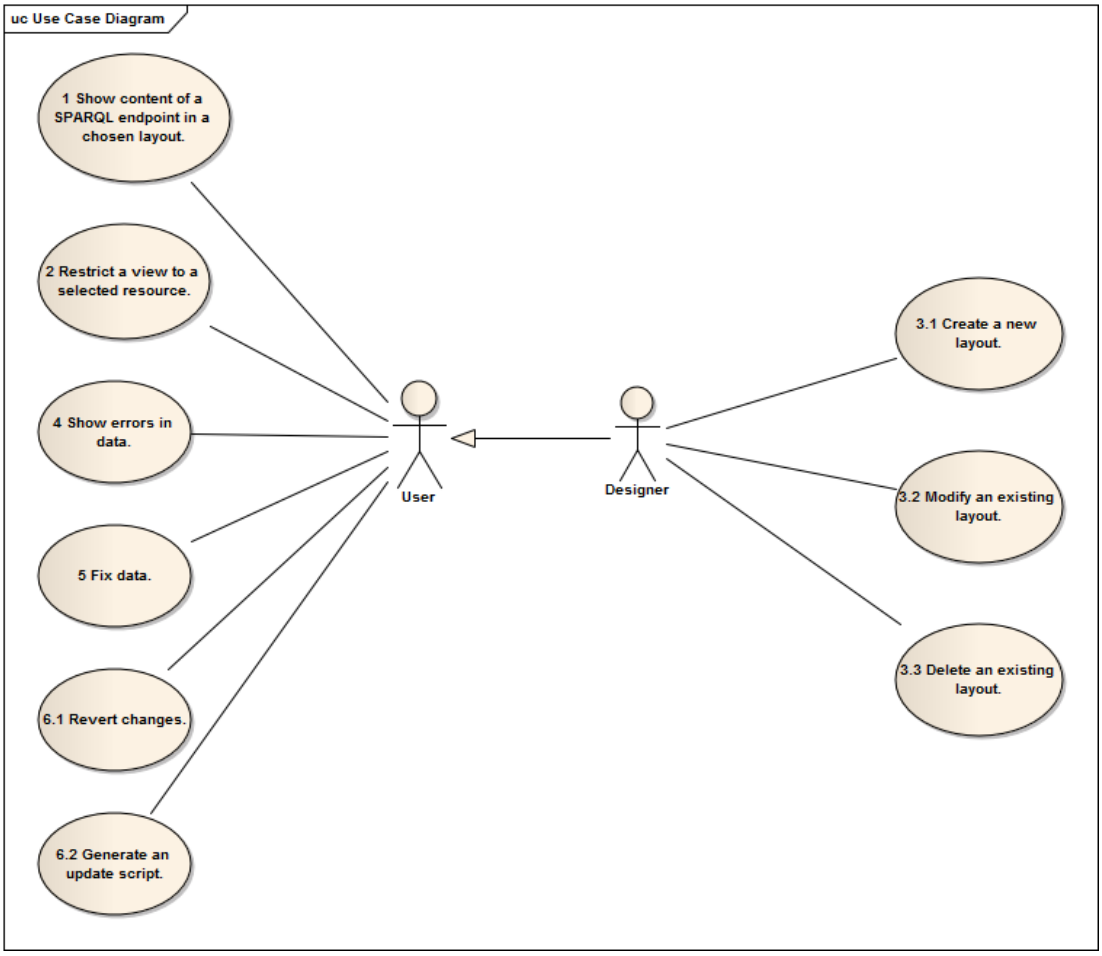

Figure 1.5: Use case diagram

All previous requirements are summarized in the following figure using a UML requirements diagram. Particular requirements are numbered for easier referencing later in the thesis and each UC number corresponds to a section number and use cases. Requirements 2.* corresponds to the section 1.2 and to UC 2, requirements 4.* corresponds to the section 1.4 and to UC 4, etc.

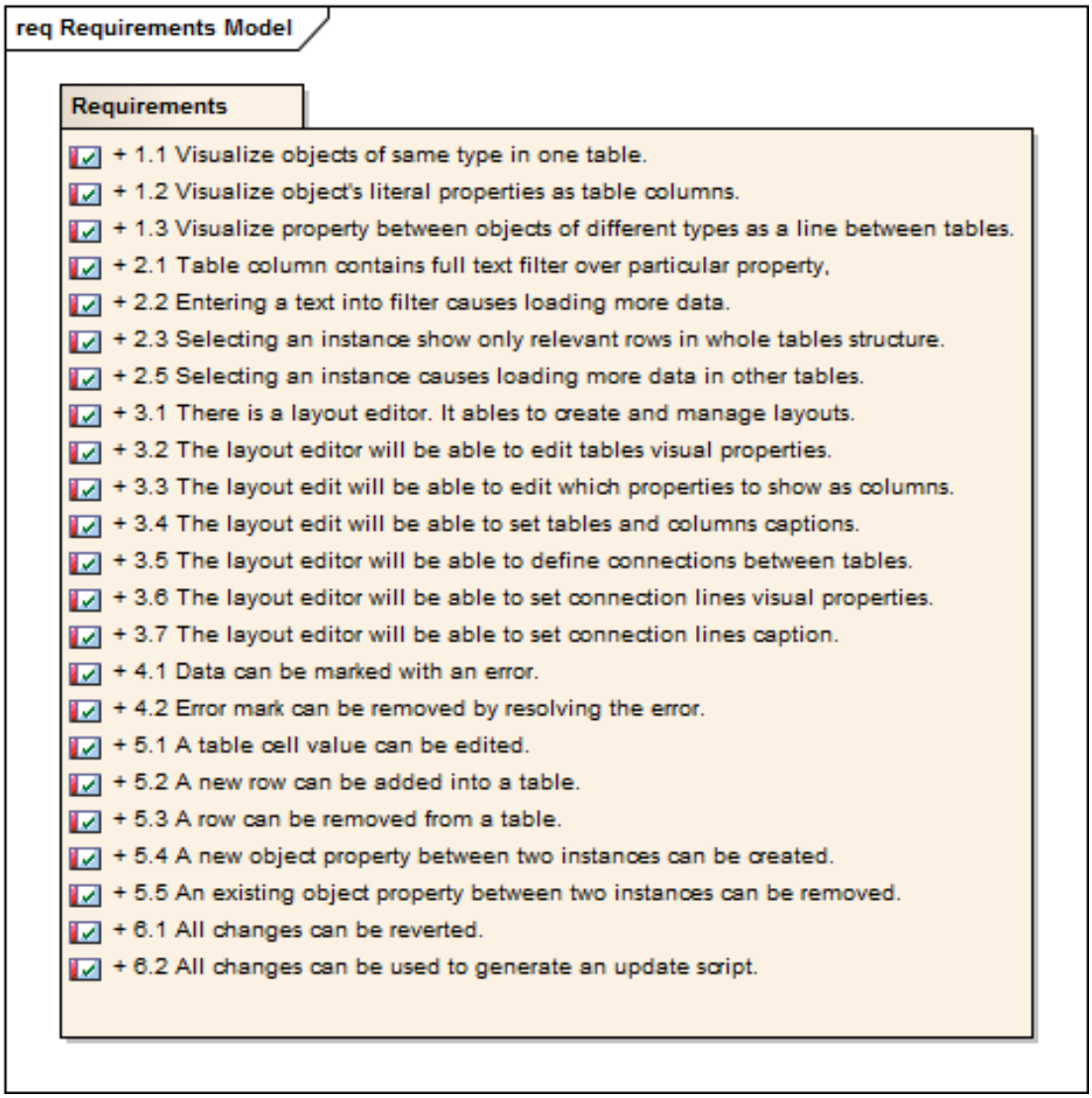

Figure 1.6: Requirements in UML.

2. Related work

There exists many tools working with RDF. In this chapter we describe some of them, show their advantages as disadvantages.

2.1 BioPortal

[11] BioPortal serves as an ontology or a RDF browser and it contains world's largest biomedical database. It provides searching of resources by fulltext or listing its content by categories. The mechanism it uses to visualize RDF data is showing them in a HTML template and a RDF data browser. Its main advantages are well treated search engine and the size of data. However this application is read only, it doesn't allow to modify data. The result is visualized in a constant template so a user can't define output. These are main disadvantages of this application.

2.2 OpenData.cz

[14] The OpenData.cz project is trying to build an open data structure in Czech Republic. It contains a data catalog where data can be searched and browsed and some extra applications used to visualize a RDF content. This project is massive in provided functionality. RDF data can be viewed in many ways, such as custom visualization via the external application Payola, visualization into a HTML template. The applications performs queries over RDF data and visualize result transformed into their data model.

2.3 OntoStudio

[12] OntoStudio allows to create and edit RDF data. This application is used to manage ontologies. The mechanims used to visualize RDF data is a data browser. It visualizes pure RDF data as a graph structure. Because of that this application can only be used by users that understands RDF well.

2.4 Protégé

[13] The Protégé application provides similar functionality as the OntoStudio. It's main purpose is to mage ontologies, as well. It visualizes RDF data in a graph structure by browsing resources. Because of that this application can only be used by users that understands RDF well.

2.5 Conclusion

As we see, the visualization of RDF into table structure is not very common. Other applications usualy visualizes RDF in HTML tables or graph structures. The big advantage of the thesis is the uniqueness of the visualization mechanism.

Moreover, other application usually requires the knowledge of RDF technologies. The thesis can resolve a situation when data have to be accessed by users which doesn't understand RDF technologies.

3. Design

3.1 Components

The architecture of the application will be based on classic MVC architecture. Main components will be Frontend and Backend. Frontend will be used as a view and Backend will contain a model and a controller. Communication between the Frontend and the Backend will be performed via web services which is the Three-tier architecture. To describe both MVC architecture and a usage via web services, the component diagram will contain components from both MVC architecture and web services point of a view.

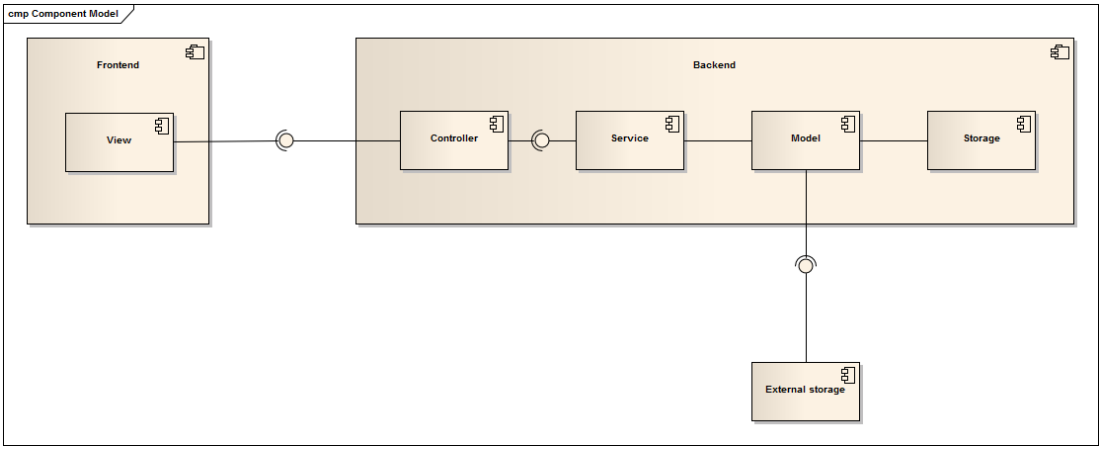

Figure 3.1: Component diagram

Whole communication between frontend and backend will flow through web services. The Controller will be used only for handling requests. It will catch concrete requests and calls relevant method in the Service component. The Service component will perform business logic operation on the Model component and return model data to the Controller which return result back to the Frontend. From the MVC point of a view, the Controller and the Service component together makes the controller. From the web services point of a view, the Controller and the Service makes a service layer or a business logic layer which communicates with a data layer.

A model will consist of the Model component and the Model component will communicate with the Storage and the External storage. The application will use its own storage for persisting non-business logic data and external storage from which the business logic data will be obtaining. Communication with external data storage will be implemented via SPARQL queries, because they can be only RDF storages. Both internal and external storages will be a part of a model from the MVC perspective, or a part of data layer from the web service perspective.

3.2 Deployment

We've decided that the frontend will be written in HTML 5 using AngularJS 1 framework. The frontend framework choice is AngularJS, because in these days

it is the most used JavaScript framework and to opposite to AngularJS 2 there are more libraries and support available for AngularJS 1. The backend will be written in Java 7. We won't use Java 8 because it is not released as stable at the moment.

As the Frontend and the Backend are different components, they should run on different servers. The part of the Backend is the persistence Storage. Which will run on some storage server. Because whole application concerns RDF, we decided that the persistence storage will be a RDF as well. There are no special requirements on servers, we can choose servers at will. For both frontend and backend server we chose the Apache Tomcat server, because it is a massively used server, can serve application of this size and is easily set and deployed. Other types we could choose are for example JBoss, NetWeaver, Glassfish, WebLogic, or others commonly used Java servers.

For internal RDF storage we'll use Apacha Jena Fuseki server because of easy integration with Java applications. SPARQL endpoint may run on different types of RDF storages so the connector to SPARQL endpoint must handle common types of endpoint.

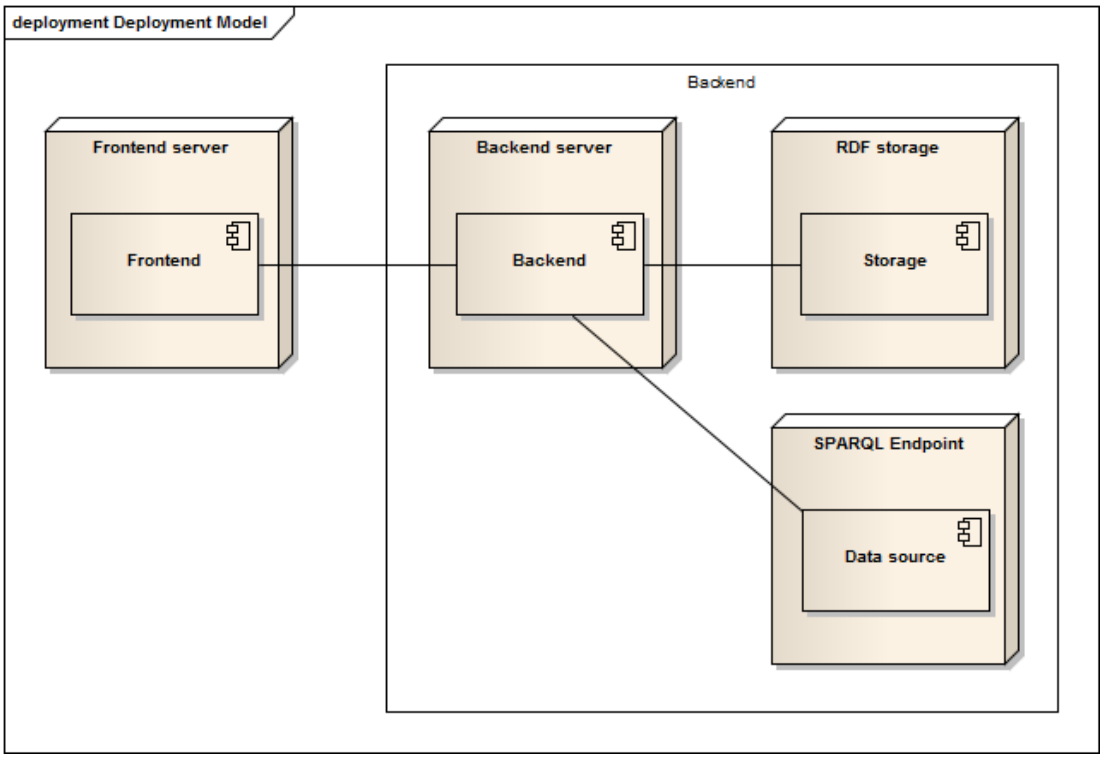

Figure 3.2: Deployment diagram

3.3 Basic requirements and functionality

RDFS includes property called "rdf:type" which defines the type of an object (its domain is 'rdfs:Class') and this property will help to group objects by type. Moreover, grouping by type is logical and easily understandable. Each group of objects can be visualized by a table. That means that a table represents a type and will contains only objects of a same type. A table consists of rows where each

table row represents an instance of type represented by the table. Each table row consists of columns where each column represents object's property, property "rdf:type" is excluded because it's already represented by the table. Name or title of a property is shown in the column header. If there is a property which object has a type which is already represented by other table, this property will be shown as a line connecting two tables.

The following figure illustrates a situation with objects of three types (a city, a country and a currency). The table "City" has three objects (or three instances of that type) with three properties (a name, a population and an area). The table "Country" has two objects with three properties as well (a name, a language and a population). The table "Currency" has two objects with two properties (a name and a symbol). Moreover, the table "City" is connected to the table "Country" with a line labeled as "lies in". That means that objects of table "City" has properties, which object has type "a country" and they are located in a table "Country". Vise-versa, Table "Country" is connected with table "City" with the line "capital", which means that some objects in table "Country" has property which object has a type "a city" and is located within the table "City". In a similar way, the label "Country" is connected to the table "Currency" and that means that some objects in table "Country" has a property which objects lies within the table "Currency".

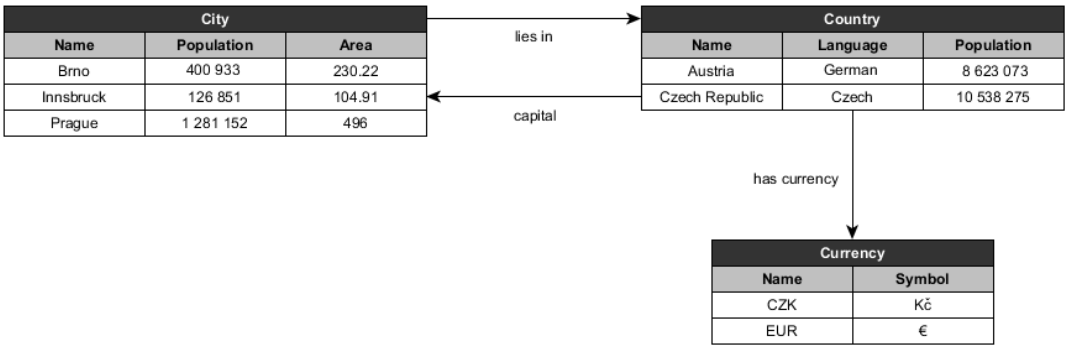

Figure 3.3: Structure of designed visualization

The important thing is that the visualization only shows data structure and will not perform any kind of data transformation. A table will strictly contain only instances of a same type and it's properties. Connection between two types is visualized via a connection between tables. A table may never contain instance of another type or a property of another type. In the example above, there will be no possibility to show column "Currency symbol" in the table "Country" as lines "Czech Republic has currency symbol CZK" and "Austria has currency symbol EUR". These information can be visualized only by connection between tables "Country" and "Currency" and after clicking a concrete instance (for example "Austria") in the "Country" table in the table "Currency" only instances related to the selected one in "Country" table will be visible (in this example "EUR").

3.4 Data loading, searching and filtering

As we noticed, a line connecting two tables visualizes a relation between two types (because table represents a type) but it represents a relation between type's objects (or table rows). It is a good way to visualize the overall preview and some connection between two types, but it doesn't represent concrete relation between two objects. In the figure above, there was a connection between a country and a city, but there was no information about objects relations. The elegant solution for this situation is, that after clicking a row table (in other words, by selecting a concrete instance) in all other tables connected to the selected one its content will be filtered and will contain only objects in a relation with selected object in the selected table. After this action (clicking the table row) the appearance will be same but there will be visualized only concrete objects. For sure there must be a possibility to cancel the selection to display the original situation. For example, by a clicking the selection again. Description of this action by words may be unhandy, better explanation provides following figure. It's meaning it straight-forward, so there is no need for an explanation.

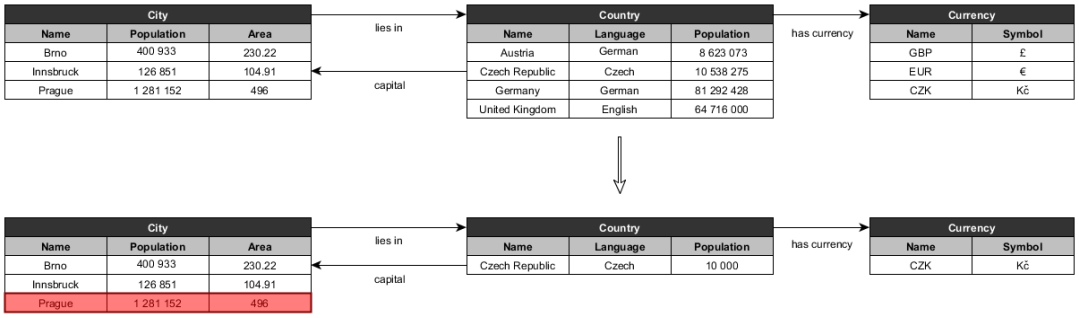

Figure 3.4: Principle of showing properties for a concrete object

There may be a need to do this operation over all tables in a schema if a user wants to propagate the filter from a start table to a target table but have not so much information about tables between. However, this operation may by very slow in a case of many tables and for that case there should be a setting to disallow this feature. But by default a propagation through whole structure will be enabled. To reduce a number of called queries we have to design an algorithm that solves the situation.

Filtering of already existing instances this algorithm:

1. Input: List of selected instances L(T), table T.
2. Mark table T as visited.
3. For each table A connected to T as a target table:
   (a) List of showed instances L(A) is empty.
   (b) For each instance X of A:
      i. If one of the selected instances from L(T) is connected to X, show it and add X to L(A).
      ii. Else, hide it.

(c) For each table V connected to A:

i. If V is marked as visited, continue.

ii. Else, repeat algorithm recursively for L(A) and V.

4. For each table B connected to T as a source table:

(a) List of showed instances L(B) is empty.

(b) For each instance X of B:

i. If one of the selected instances from L(T) is connected to X, show it and add X to L(B).

ii. Else, hide it.

(c) For each table V connected to B:

i. If V is marked as visited, continue.

ii. Else, repeat algorithm recursively for L(B) and V.

The initial input is only one instance but in that case it's not difficult to transform it into a list to make the algorithm work for the initial case. Test, if "one of the selected instaces L(T) from the table T is connected to X in table A" means:

- **If A is a source table**: There is an connection between the table T and the table A and one of instances in L(T) has an object property which target is the instance X.
- **If A is a target table**: There is an connection between the table T and the table A and the instances A has an object property which target is one of the instances from L(T).

This algorithm may also serve for loading new data. Before the steps 3.1 and 4.1 we can add a new step for loading new data. Its input will be current table - the A or the B in this case, a table which invoked the algorithm - the table T in this case, and a list of selected instances L(T). Based on this input a query will be generated that loads new data for the table A (or the table B) limited for the selected instances. The query will have the following form:

```
\# Selecting instances which are targets of the selected one.
SELECT ?instance
WHERE {
   VALUES ?value { :val1 ... :valN }
   ?instance property ?value .
}

\# Selecting instances which are sources of the selected one.
SELECT ?instance
WHERE {
   VALUES ?selected { :val1 ... :valN }
   ?selected property ?instance .
}
```

This quere selects all instances which property objects are values from the list val1 ... valN. Those values are URIs of instances from L(T). The property in the query is the the property connecting tables A and T.

3.5 Lazy loading

It is clear that a table can contain more rows that can be displayed in a table area. This situation can be solved by scrolling bars or table pagination. We've chosen the first option - scrolling bars. Both solutions have its advantages and disadvantages. Table pagination can be more adjustable - a user can specify the page where to move, jump to the first page of the last page, but it requires extra control elements which can be confusing because of multiple tables on a screen and it take more screen space. There are two issues connected with scrolling the table. The first issue is a data size, the second issue is ordering and filtering. Therefore, there will always be all instances available to satisfy current settings.

To compensate disadvantages of scrolling bars (user can't choose exact data block to show) there must exists filter and order methods. As usual in other web applications, by clicking a table header, the table rows will be sorted by clicked column in descending order. After second click table rows will be sorted by clicked column in ascending order. After third click the order will turn back to default. These actions can be caused by clicking the table header itself or by clicking added arrows to control the order. We've choose not to show arrows and use the first option to minimize the amount of control elements. For filtering table rows, a table header will contain a text field and after entering a value into text field, table rows will be filtered by appropriate column by fulltext search. Note that there may be an issue with filtering within one table and between two related tables. The application have to handle this situation. That means, that filtering and ordering within a table have to be possible even if content of tables is restricted only to show concrete instances.

A table may content a large amount of rows. To improve the loading time of rows, the data will be loaded in parts. At first, there will be loaded only amount of rows to fill the table size. There is a chance that a user won't need the rest of the data. If a user will scroll the table, only in that situation the next part of data will be loaded. It's a common technique in web applications. It is clear that a table and its adjaced ones may not contain all existing instances. Because of that, after each filtering action (filtering by text search and filtering by selecting an instance) the tables data have to be reloaded and refill with new ones satisfying the filter. That applies for all adjaced tables in case of selecting an instance.

The important thing is that an object property have to act like bidirectional although its defined like unidirectional (subject is a source and object is a target). That implies that selecting an instance have to work even in a case that the table where the selected instance lies is a target table of some property, not a source.

3.6 Layout editor

The layout editor tool is demanded by requirement 3.1. This tool will be used for defining a static structure to display data. In a short, in this editor a user adds some tables, adds a columns for each table and then connect some tables. This result will be saved and will be used for visualizing data. First part of the layout editor is managing layouts. There will be a list of already saved layouts. A designer can create a new one by clicking a button or edit already existing layout by clicking its name. Both of those action opens an editor where a layout can be

modified. In the list of layouts there also should buttons for removing layouts.

First step in defining a layout is to create a new table. After clicking a button a small rectangle is created in a working area. This rectangle represents a table in the layout. More specific properties of a table represented by a rectangle can be edited in a modal window which is opened by double clicking a rectangle. The most important property to set is table's class. That means that table will contain only instances with that defined class. It should whole URI of the class and namespace prefix can be used as well. It would be useful to define some visual properties for a table. The following list defines all visual properties that will be editable for a table.

Position of a table on a screen This can also be done by drag and drop a rectangle representing the table.

- Size of a table (width and height)
- Size of a font in a table
- Background color
- Color of borders
- Thickness of borders
- Padding in table cells

Editing of these settings satisfies requirements demanded by the requirement 3.2. For a table its caption can be set as well as caption of individuals columns to satisfy the requirement 3.3. Captions and labels will be described more precisely in a separate section. Last things to set for a table are its columns. In a model window for a table there should be a column list with possibility to add a new one or to remove an existing one. A column may be defined for different kinds of data:

URI URI a simple URI of an instance.

Label Label is a simple name of an instance.

Other Display any other demanded property.

Lines between tables will be created by clicking a button after which a modal window will appear. By clicking the source table and then clicking the target table and defining a property connecting them we define a connection. This connection describes that there is an object property which subject's domain is from source table class and its object's domain is from target class. Then a line connecting rectangles will appear. After double clicking a created line a modal window will appear again. Instead of mentioned properties (source table, target table, property) some visual properties will be available to set for a line:

- Line thickness
- Line color

- Label font size
- Label font color

The start and end point of a line will be generated automatically by position of related tables. Availability of those visual settings will satisfy the requirement 3.6. To satisfy a the requirement 3.7 as well a caption of a line have to be editable as well.

3.7 Layout dictionary

Because the whole thesis deals with RDF, we want to store layouts in RDF as well. We have to define a dictionary which defines how to store a template in RDF. The used namespace will be "http://mff.cuni.cz/vkgmt#". "vkgmt" is an abbreviation from the title of the thesis - "Visual knowledge graph management tool". In this dictionary there will be 4 classes to describe different type of layouts:

- **vkgmt:ScreenLayout** - for representing a layout as a complex.
- **vkgmt:BlockLayout** - for representing a table properties.
- **vkgmt:ColumnLayout** - for representing a column properties within a table.
- **vkgmt:LineLayout** - for representing a line connecting two tables.

vkgmt:ScreenLayout

Class ScreenLayout will holds a list of block layouts, a list of line layouts, a layout name, a list of namespaces and a propagation settings. Those data will be described by following properties:

- **vkgmt:blockLayout** - its a list of all block layouts in the layout and its domain is a list of vkgmt:BlockLayout types.
- **vkgmt:lineLayout** - its a list of all line layouts in the layout and its domain is a list of vkgmt:LineLayout types.
- **vkgmt:namespace** - its a list of all namespaces in the layout and its domain is a list of strings in a format "prefix":"namespace".
- **vkgmt:filterPropagation** - describes if allow or disallow a filter propagation throug whole schema. Its domain is one of those values: NEIGHBOURS, ALL.

vkgmt:BlockLayout

Class BlockLayout will holds data about a table. It contains properties describing a list of columns within the table, table type, background color, height, width, position, label source, label type and label language, font color, font size, border color, border style and border thickness:

- **vkgmt:background** - String value of which color the table background will have.
- **vkgmt:fontColor** - Color of the whole text within the table.
- **vkgmt:fontSize** - Size of the whole text within the table.
- **vkgmt:forType** - Describes for which RDF class the table is designed.
- **vkgmt:height** - Height of the table on a screen.
- **vkgmt:labelLang** - Describes the language of the table caption.
- **vkgmt:labelSource** - Describes the source of the table caption, if the label type is set to CONSTANT or PROPERTY.
- **vkgmt:labelType** - Describes what resource makes a table caption. Its domain is one of those values: URI, LABEL, CONSTANT, PROPERTY.
- **vkgmt:left** - Describes the X position (horizontal) on a screen.
- **vkgmt:lineColor** - Color of borders within the table.
- **vkgmt:lineThickness** - Thickness of borders within the table.
- **vkgmt:lineType** - Type of borders lines within the table.
- **vkgmt:top** - Describes the Y position (vertical) on a screen.
- **vkgmt:width** - Width of the table on a screen.

vkgmt:ColumnLayout

Class ColumnLayout will holds data about a column within a table. Some of properties are similar to those contained in the BlockLayout. It contains properties describing a used aggregate function, label source, label type and label language and a property designed for the column:

- **vkgmt:aggregateFunction** - Describes which aggregate function to use in a case of multiple values.
- **vkgmt:labelLang** - Describes the language of the column caption.
- **vkgmt:labelSource** - Describes the source of the column caption, if the label type is set to CONSTANT.
- **vkgmt:labelType** - Describes what resource makes a column caption. Its domain is one of those values: URI, CONSTANT.
- **vkgmt:property** - Describes which RDF property is assigned to the column.

vkgmt:LineLayout

Class LineLayout will holds data about a line connecting two tables. Soma properties are similar to those contained in the BlockLayout or the ColumnLayot. It contains properties describing a list of columns within the table, table type, background color, height, width, position, label source, label type and label language, font color, font size, border color, border style and border thickness:

- **vkgmt:fontColor** - Color of the line caption.
- **vkgmt:fontSize** - Size of the line caption.
- **vkgmt:fromType** - Describes the source table which means the property subject.
- **vkgmt:toType** - Describes the target table which means the property object.
- **vkgmt:labelLang** - Describes the language of the line caption.
- **vkgmt:labelSource** - Describes the source of the line caption, if the label type is set to CONSTANT or PROPERTY.
- **vkgmt:labelType** - Describes what resource makes a line caption. Its domain is one of those values: URI, LABEL, CONSTANT, PROPERTY.
- **vkgmt:lineColor** - Color of the line.
- **vkgmt:lineThickness** - Thickness of the line.
- **vkgmt:lineType** - Style of the line.
- **vkgmt:property** - Describes which RDF object property is connecting the tables.

3.8 Multiple values

A table cell contains some property of an object. We can imagine a situation where some object has a property with multiple values, for example "A rainbow has a color". Rainbow contains seven basic colors and that means in this situation, a tables cell will contain seven values. This situation can be solved by following ways:

- show the first value
- show all values as list
- use an aggregate function (minimum, average, ...)

Depending on a property value type, one of these methods have to be configurable for every table for every property.

3.9 Titles

The next issue to solve is, how to display captions or labels. This is used in defining table caption, table column caption and line label. In following list there are suggestion for different sources of captions.

- URI - This source of a caption shows simple URI.
- rdfs:label - Common property in RDFS for labels.
- dc:title - Another property describing a label.
- skos:prefLabel - Another property describing a label.
- User defined value.

In definition of types and properties a user can use a namespace prefix. The layout editor have to contain a list of used namespaces with its prefixes to construct a correct model. Checking that all prefixes are defined is not in a scope of a thesis. But it can be a suggestion for a future work.

3.10 Marking error data

Concerning the requirement 4.1, some data can be marked as possibly invalid. We were given a error describing ontology for that purpose. The given ontology is broader, but for our purposes only properties we want to use are: error triple, value, severity and description. Resource describing an error has type of daq:Observation.

Property	Error triple
URI	daq:problemDescription
Domain	a RDF statement
Meaning	Describes which property has an error. It is related to whole triple.
Property	Value
URI	daq:value
Domain	Percentage from range 0 - 100
Meaning	A value 100% means that a triple is correct. The lower the value is the bigger is the error. Therefore an error is considered only those objects that have the value less than 100%.
Property	Severity
URI	daq:severity
Domain	daq:Severity
Meaning	Severity of the error. Can be one of the following: error, warning, info.
Property	Description
URI	dcterms:descrption
Domain	string
Meaning	A human readable description of the error.

Namespaces used in definition of error ontology:

- daq - http://purl.org/eis/vocab/daq - Dataset Quality Vocabulary
- dcterms - http://purl.org/dc/terms/ - DCMI Metadata Terms

The following example shows an error for the statement
'The city ¡http://example.com/Prague¿ has a total population of I268S1'.
This property is clearly wrong and that is described by the error.

```
<http://example.com/Prague> example:population "I268S1" .

_:node1 a rdf:Statement ;
rdf:subject <http://example.com/Prague> ;
rdf:predicate example:population ;
rdf:object "I268S1" .

ex:obs1 a qb:Observation .
ex:obs1 daq:problemDescription _:node1 .
ex:obs1 daq:severity daq:error
ex:obs1 daq:value "0.3"^^ .
ex:obs1 dcterms:description "Error - 'IS68S1'
  is not a valid numeric value for." .
```

Because the user vocabulary describes error for a RDF triple, there can be only these two types of errors to visualize: literal property or an object property. Error in a literal property means, that there is a problem with a table row. In this case a mark should be shown within the table column. In the other case, error in an object property, the situation is difficult because this error can affect three object - a source table, a target table and a line connecting them. Marking the property object isn't suitable because this object may exist in other table and is perfectly correct, error is only in the object property therefore marking a property object would be misleading. A next way is to mark a line connecting tables as error. The problem is that it could invoke a feeling that whole connection between tables is wrong. The last way to display the error in object property is to mark the subject of the property. That means to mark a table row. Because marking the table cell is already used for a literal value error, a mark for an object property error should be in some special position - for example next to a table row or on the edge of a row.

Because an error can have three degrees of severity, the mark should visualize all three degrees. But a value of en error have to be visualized as well. The solution is to identify the severity by the mark shape and the value by the mark color. There are no special demands on shapes and colors so we can design them at will. The following list defines an error mark appearance by error severity:

- Error severity - square shape
- Warning severity - triangle shape
- Info severity - round shape

And the following list defines an error mark appearance by error value:

- down to 90% - light green background
- 90% - 75% - yellow background
- 75% - 50% - orange background
- 50% or less - red background

There is one property left for an error definition - a description. This description will be shown after mouse over an error sign. An error description may be large and it would be confusing to show it directly. Instead of that, after mouse over there appear a popup text box with the error description. After a mouse moves away, the description disappears.

In the following figure there are examples of data containing errors marked with designed marks. The population of Prague has a strange value - and it's marked by an error mark with the warning severity and value about 50 - 70%. After a mouse over the sign an error description appeared. The second literal value is in a cell for a leader of Austria. Clearly this value is incorrect and there is an error describing it. There is an error sign with error severity and value under 50%. There are also two marks that says an object properties are incorrect. They have both the info severity, one has the value down to 90% and the second one about 75 - 90%. Errors are there because both cities (lies in Czech Republic) has object properties that says those cities lie in Austria.

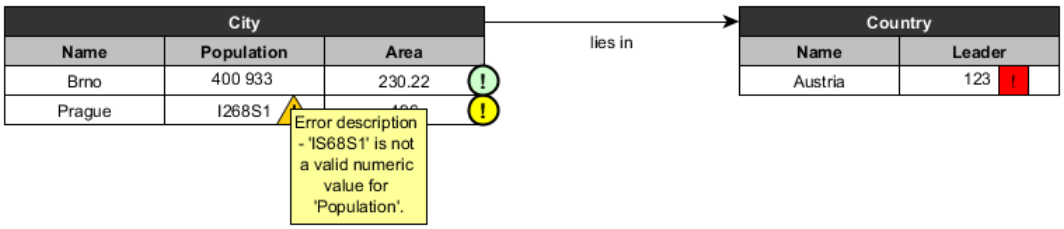

Figure 3.5: Examples of marked corrupted data

Errors fixation will consist of two steps. At first a user will modify a corrupted value (modification of data is described in next section). The second step is to mark an error as resolved. After left clicking an error mark a popup window will appear with only one button to resolve the error. After clicking it the error sign will disappear because the error is now considered as solved. There will be no changes in data structure itself, only changes in error list. Therefore, an error list should be separated from data structure to not impact each other.

3.11 Modifying data

There are 5 requirements for a data modifications. Each of them requires one type of an operation. Because clicking a part of a table is reserved for another actions mentioned before, all operations will be triggered by a context menu. Opening of context menu will be triggered by rightclicking a table cell. Rightclicking a table cell selects a cell or whole row for modifications. In the context menu there will be a list of operation to do. If an operation concerns a table cell, it will only

affect the cell, otherwise it will affect whole table row. All performed changes have to be visible in a data structure immediately.

Editing a table cell

Operation of editing a table cell is demanded by the requirement 5.1. By this operation a user can set a new value to a cell. From the RDF point of a view, it will edit a value of a property which object is an literal value. The new value can be added even if the cell was empty before. In this case a whole new value is created, not just modified the object value. For more user friendly interface there will be a new operation for removing the cell value. The difference is that it doesn't set a value to empty string as it could be done by editing, but it will remove the property. From the SPARQL point of view, an statement for editing a value would be

```
DELETE <uri> property 'old-value';
INSERT DATA { <uri> property 'new-value' };
```

but for removing a cell it would be only a first statement - there would be no need for inserting an empty value.

Adding a table row

Operation of adding a new row into a table is demanded by the requirement 5.2. From RDF point of a view it creates a new resource with a new URI and attaches some properties to this resource. To make this operation successful a new URI must be entered. This is only mandatory value to add, but there must be a possibility to add values for all other properties defined for the table, but there won't be mandatory. A new URI will be entered by a user, it won't be generated automatically, but a check of a correct URI have to be done. A type of the new resource is defined by the table where the operation was invoked and because of that a user don't have to enter a new instance's type.

Removing a table row

Operation of removing a row from a table is demanded by the requirement 5.3. From RDF point of a view it removes properties for some resource. However a URI of a resource itself will not be deleted, because only properties for a resource will be removed. Note that only properties listed in a table should be removed, because a data source may contain others properties that are not listed in a table and they must remain untouched.

Adding an object property

Operation of adding a new object property is demanded by the requirement 5.4. Its purpose is to add a connection between two instances. From the RDF point of a view it means adding a property which both subject and object are resources. To make this operation valid, after selecting this type of an operation a popup window will be shown. The popup window will contain a list of object properties

that are defined for the selected table. That means it shows names of all tables connected to the selected one. After choosing a property to add a values for the selecting property will be loaded from target table. This will be the only way to add an object property and because of that the result will be valid. Because a user can only add an existing loaded instance for a defined existing property.

In the structure described in the figure below we can show how the process will work. In a table labeled as 'Type1' a user wants to add a new object property for a row 'selected_instance_1'. After showing the popup window a list of properties 'prop1', 'prop2' and 'prop3' appears to select, because these are properties connecting a selected table to other ones. After selecting a 'prop1' property to add, a list of 'type2_instance1', 'type2_instance2', and 'type2_instance3' instances will appear and a user will chooses one of them to add. Similarly, if a 'prop2' is selected, instances 'type3_instance1' and 'type3_instance2' will appear. And in the last case, if a user selects a 'prop3' properties, one the 'type4_instance1' instance will appear to add.

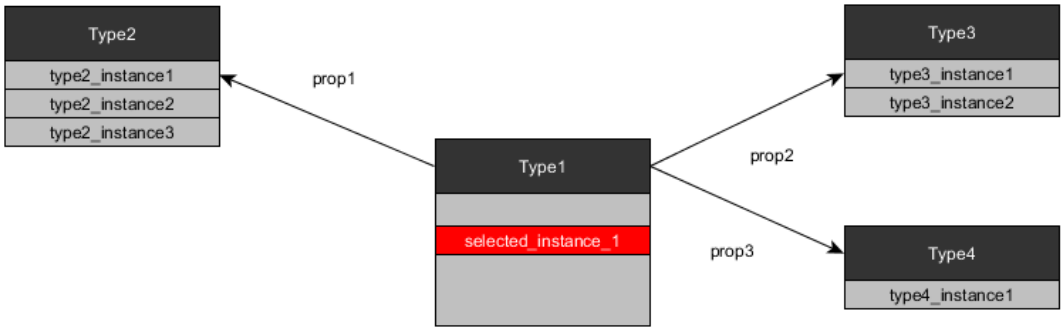

Figure 3.6: Example data for adding an object property

Removing an object property

Operation of removing an object property is demanded by the requirement 5.5. Its purpose is to remove a connection between two instances. From the RDF point of a view it means removing a property which both subject and object are resources. To make this operation valid, after selecting this type of an operation a popup window will be shown. The popup window will contain a list of all bounded instances from other tables. Each of those instance can be marked for a removing and be removed at once.

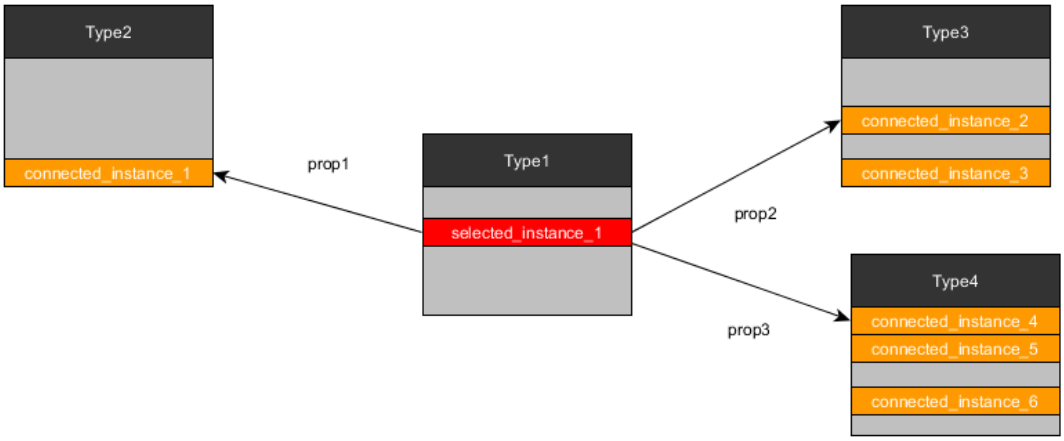

Figure 3.7: Example data for removing an object property

In the structure described in the figure above we can show how the process will work. A user wants to remove an existing object property for an instance labeled as 'selected_instance_1' in the table labeled as 'Type1'. We can see that the selected instance is connected to instance 'connected_instance_1' in table 'Type2' via property 'prop1', with instances 'connected_instance_2' and 'connected_instance_3' in table 'Type3' via property 'prop2' and with instances 'connected_instance_4', 'connected_instance_5' and 'connected_instance_6' in table 'Type4' via property 'prop3'. After the popup window appears there will be a list of 6 objects properties to remove. It will be displayed as a list of pairs of property : instance to show to user which instance via which property is connected to the selected instance. In this it will be following ones:

- prop1 : connected_instance_1
- prop2 : connected_instance_2
- prop2 : connected_instance_3
- prop3 : connected_instance_4
- prop3 : connected_instance_5
- prop3 : connected_instance_6

3.12 Handling changes

All changes performed on data have to be stored in some temporary structure and must not affect the original data model. Only after saving changes they will be stored to the original data model and list of changes will become empty. This concerns only saving changes into data model used in the application. But persisting changes into a data source is not in a scope of the thesis. Saving changes only affects data loaded in the application. To perform the persist a user have to generate an update script and run it on a data source by itself. Generation of an update script is demanded by the requirement 6.2. For modifying RDF data is used a SPARQL endpoint which modifies data in some RDF data storage. It

uses the SPARQL language and because of that the update script have to be generated in the SPARQL language. The SPARQL update script will be further referenced just as "update script".

An update script will be generated into a text area where it can be copied by a user. But as a script may be quite large for big amount of changes, near the text area there should be a button for downloading the script in the text area as a file. Result file will be a simple text file it won't be a runnable script. For a better readability of a script it should be formatted in a human readable form. There are no special demands for the formatting, it just should have a subjectively good formatting. It is unwanted to generate a script which whole content consist of only one line without any indentation.

An algorithm for generating a script will be simple. It will cycle through all changes and for each change it will generate a simple subquery. This algorithm ensures that changes done by a script will be exactly the same as done in the application because it doesn't do any reordering or optimizations. There are 6 types of changes that can be done. In the following table there are listed all types of changes, described what does the change cause and what subquery should be generated.

Operation
Impact
Query
	Editing a table cell
	Update a property value
	
	`DELETE <uri> property 'old-value';`
	`INSERT DATA { <uri> property 'new-value' };`

Operation Impact Query	Removing a table cell Remove a property value `DELETE <uri> property 'old-value';`

Operation
Impact
Query
	Adding a table row
	Create a resource with all properties
	
	`INSERT DATA {`
	`<new-uri> prop1 'val1'; ... propN 'valN'`
	`};`

Operation
Impact
Query
	Removing a table row
	Delete resource properties defined in the table
	
	`DELETE {`
	`<uri> prop1 'val1'; ... propN 'valN'`
	`};`

Operation
Impact
Query
	Adding an object property
	Inserting a property
	
	`INSERT DATA {`
	`<source-uri> property <target-uri>`
	`};`

Operation
Impact
Query
	Removing an object property
	Removing a property
	
	`DELETE {`
	`<source-uri> property <target-uri>`
	`};`

Note that in case of removing a table row there have to be entered all properties defined in table, because the resource in a data source may contain other properties that aren't shown in the table and running of simple

```
DELETE { <uri> ?p ?o };
```

would cause removing of all resource's properties not only ones removed by a user.

It is clear that such a script may contain unnecessary or duplicate operations and thus won't be optimized. In the following figure there are examples of scripts that can be optimized.

```
# First example
INSERT DATA {
<http://example.com/uri1> dc:description 'A description'
}
DELETE {
<http://example.com/uri1> dc:description ?o
}

# Second example
INSERT DATA { <http://example.com/uri1> ex:prop1 'value1' }
INSERT DATA { <http://example.com/uri1> ex:prop2 'value2' }
INSERT DATA { <http://example.com/uri1> ex:prop3 'value3' }

#Third example
DELETE {
<http://example.com/uri1> dc:description 'A description'
}
INSERT DATA {
<http://example.com/uri1> dc:description 'A description'
}
```

Figure 3.8: Examples of unoptimized SPARQL scripts.

In the first example there is an update script that could be created for example by editing a property 'dc:description' at first and then followed by removing whole object by removing a table row. The script won't have to be run at all, because in second statement it removes a property inserted in a first statement. In the second example all three insert statements could be merged into just one insert statement. In the third example the statements don't have to run at all because the second statement inserts a property that was removed by the first statement. Those was only simple examples and in more complex cases an optimization could be even more sophisticated. An optimization may cause some problems. For example in the first example the first statement only inserts one property but the second statement removes all 'dc:description' properties and thus it can't be optimized be simple canceling of both statements. As said, in more complex examples there can be complex relations which optimization is not trivial. But optimization of an update script is not in a scope of the thesis and is not demanded to be implemented. However, it can be a suggestion for a future work.

3.13 Reverting changes

The requirement 6.1 says that all temporary changes done in data model must be able to be reverted. There will be a button and after clicking it all temporary

changes will be discarded and data model will return to its original status. There can be a conflict in following case. A content of a table adjusted by loading more rows into table by some filter operations (selecting an instance in a neighbor table or entering a text in a column search/filter field). This new data are considered an original data model. Now a user perform some changes to the data model. After performing the changes mode rows are loaded into table with data changed by a user. In that situation data newly loaded into table (but not edited by a user) should be consider as the original data model, but changes performed by a user should be temporary. After discarded changes the modification performed by a user have to be reverted by data loaded into table by filters operation - in both the first and the second wave have to remain.

3.14 Class diagrams

At first we'll define data objects for the data model. Basically there are two types of objects. First one describes layout definition. The second one describes business data.

Because layout object can be persisted into RDF storage, it have to contain URI and type. URI serves as unique identifier of the object and type describes object class. There will be common parent class containing only these two properties and will be abstract because it can't exists on its own and its name will be RdfEntity. There have to be an object that hold whole layout definition. We'll call it ScreenLayout. It will contain property "name" because a layout should be named for better orientation. Also, it will contain two collections of sub-layouts - one that describes tables and one that describes connection between tables. The first one is BlockLayout and the second one is LineLayout. And at least, it'll contain about used namespaces - it will be a map which keys are namespace prefixes and values are full namespace identifiers.

BlockLayout contains a set of properties describing table appearance (colors, font, position, size). More interesting are properties describing data structure. Property "forType" describes for which types of RDF objects is this table defined. It is a value of "rdf:type" property and basically it defines type of all instances within the table. The pair of properties "titleSource" and "titleTypes" defines, what the table caption will be. TitleType defines how the caption will be loaded and titleSource describes the source from where it should be loaded. The last property "properties" contains list of properties that should be visible for instances displayed as table columns. Each of these properties has its own object called RowProperty. The RowProperty object only defines which aggregate function should be used in case of multiple values and what is property name.

LineLayout contains information about line connecting tables and also describes data structure. Interesting property is "points" containing all points on screen which the line crosses - it can be a multiline. The property fromType describes source table and property toType describes target table. The property called "property" describes which RDF property is connecting the tables. And similarly as in BlockLayout there is the pair of properties titleType and titleSource describing the line caption.

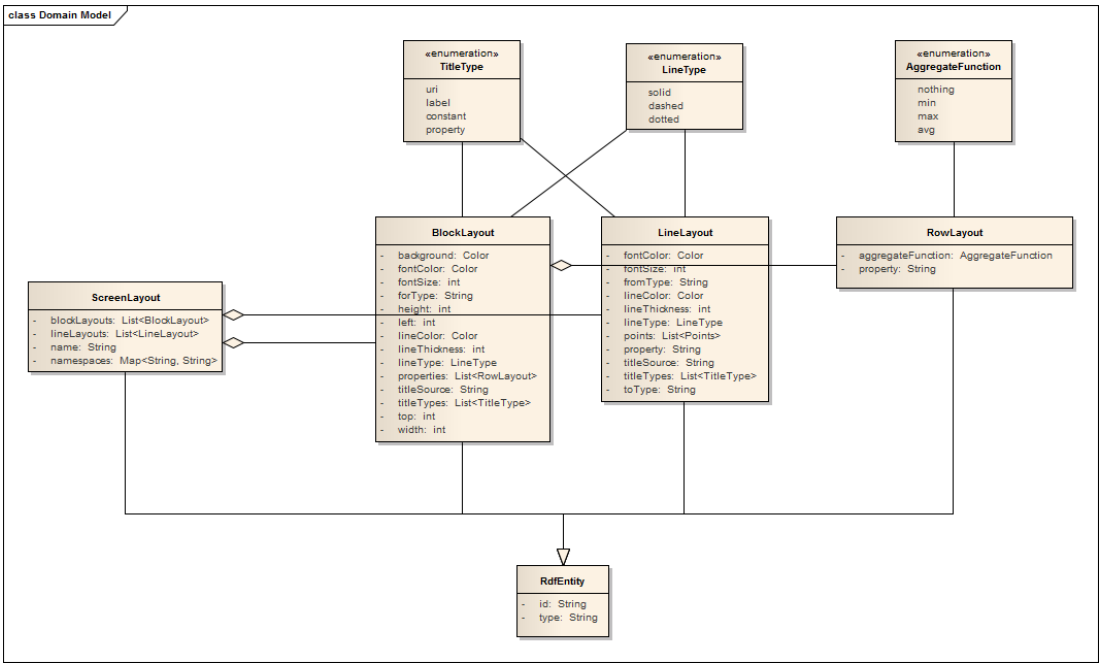

Figure 3.9: Layout class diagram

Objects of type ScreenLayout and its child elements will contain information about visual appearance and data structure. The business logic data itself will be stored within the object with name DataModel. It's a simple container that will contain a list of objects representing tables. Objects representing the tables will be named as RdfTable. This object contain a property called "forType" which represents RDF type of a table. This property will serve for matching a type to a BlockLayout object with similar property "forType". The object RdfTable also contains list of columns that should correspond with property "properties" within BlockLayout object. The difference is that BlockLayout describes how the data should look like, but RdfTable describes a real state. The last and the most important property is "instances" which represents concrete instances of a same type defined for the table. The make rows of this table.

Instance objects will be represented by class called "RdfInstance". The main property is called "instanceUri" that is a unique identifier of a instance in both data model and RDF storage where this entry comes from. Next interesting property is "type" which defines type of the instances and is same as the table's "forType" property. The class RdfInstance also contain two collections called "literalProperties" and "objectProperties". The first collection , "literalProperties", contains literal properties of the instances. The second one, "objectProperties" contains references to other instances in adjacent tables. Reference between two instances of different types is identified by their instance URIs and a property that connects them.

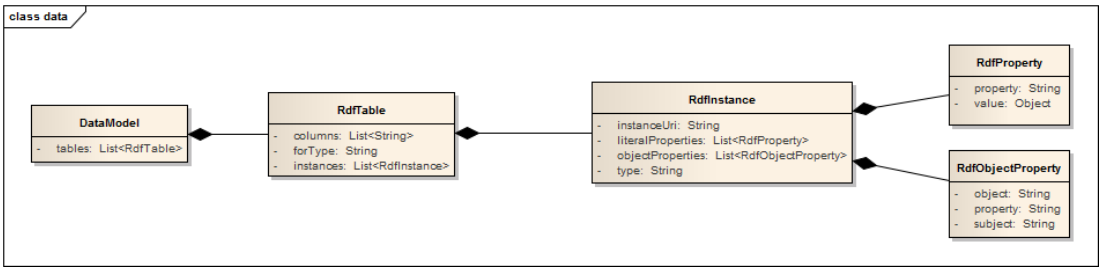

Figure 3.10: Business data class diagram

Note that there are properties with duplicate values. For example, a property "forType" in "RdfTable" contains same information as "type" in "RdfInstance" or property "object" in "RdfObject" contains same information as "instanceUri" in "RdfInstance". Properties like these are there for easier searching of dependencies or other queries in frontend.

Those were classes used as data containers. Follows the classes and interfaces within the service layer. We can see that according to Java best practices, each service class has an interface that they implement. For example, there is an interface LayoutController which is implemented by class LayoutControllerImpl, an interface DataService which is implemented by class DataServiceImpl, etc. Also we can see methods delegating through the tiers. The design of these classes is divided to three layers - a controllers, a services and a DAOs.

DAO stands for "database access object" and in this case each means objects responsible for storing model into the persistent storage. In this design, its classes implementing the "SparqlDao" interface. Each DAO class has three methods - load, insert, update and delete. The first one serves for loading object by id. Insert method persists new object into storage and updates method persists changes on an existing object. DAO classes works on a higher level than SparqlConnector class, which they use. The generates concrete SPARQL query which is sent to "SparqlConnector" class which is used for low-level connection to a storage - it only stores or retrieves objects by simple queries.

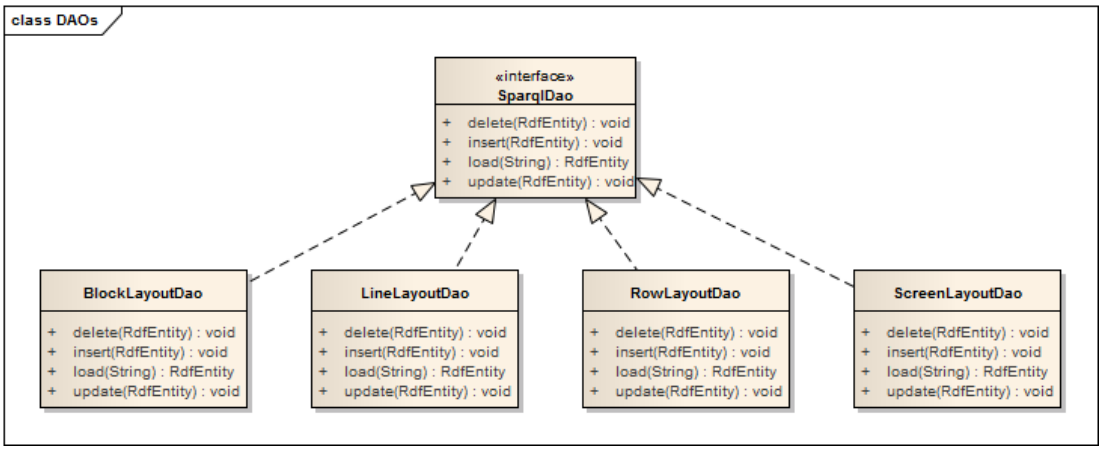

Figure 3.11: Class diagram for DAO hierarchy

One tier above, there is LayoutService interface which contains three method and only for a work with ScreenLayout object. Direct work with others Layout

object is not necessary, because they aren't used as itself but only as a part of whole layout within ScreenLayout objects. The "getLayouts" methods is used for loading all save layouts. It doesn't load subcollections as it may take a large amount of time. For loading a whole detail including subcollections, a "getLayout" method can be used. LayoutService handles the business logic for loading or saving whole objects and distributes steps to DAO objects. Interface LayoutController is just used for connection between web service request and business logic in service tier. LayoutController doesn't perform any sophisticated logic, just delegates a method call to service tier.

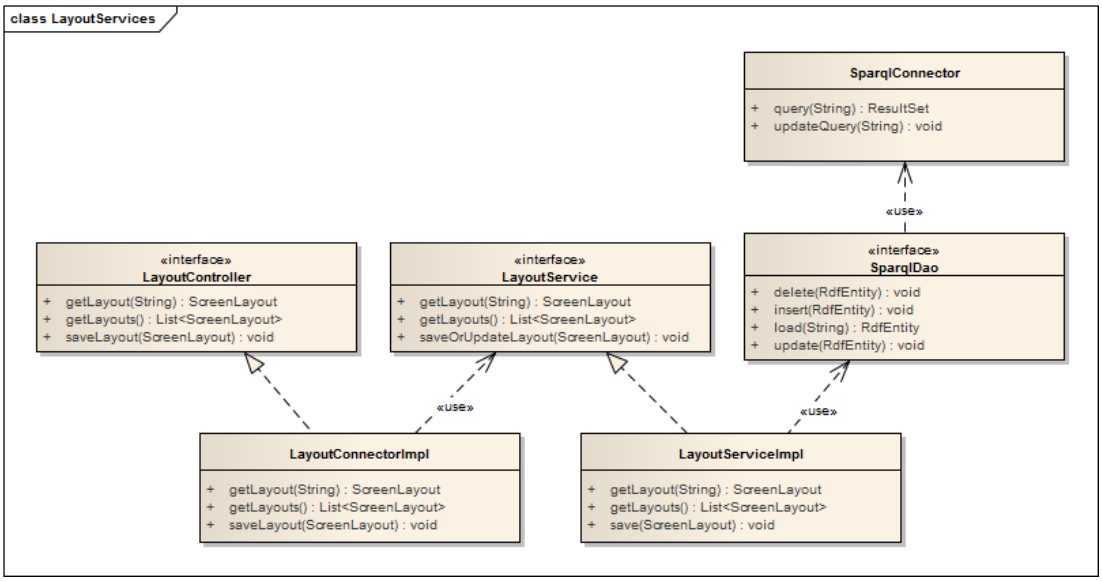

Figure 3.12: Class diagram for layout service hierarchy

Very similar is the right part of the diagram which is used for handling business logic data. The object on the lowest level - DataConnector serves for performing a SPARQL query in external storage. It contains two methods - one for loading whole defined data model and one for loading just one type instances from the model. Three different classes that implements this interface contains different function dependent on different endpoint types. For example, class VirtuosoDataConnector can use special functions for fulltext search that can be called only on the Virtuoso endpoint type and can't be called on Jena type or other types. Class DataServiceImpl decides which DataConnector to use by EndpointType parameter.

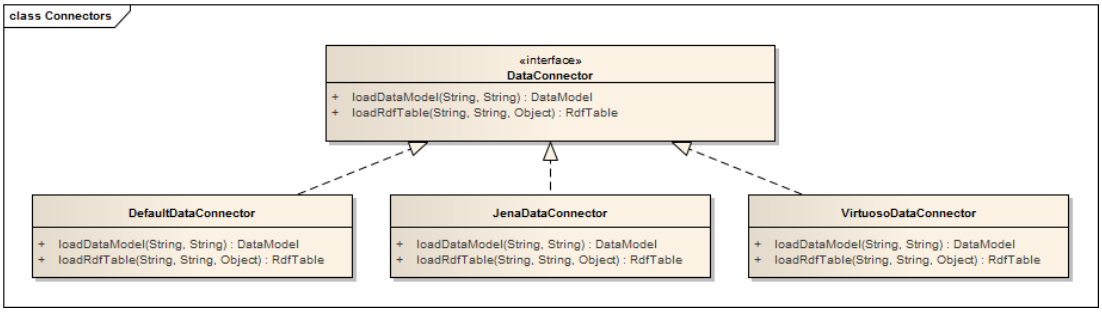

Figure 3.13: Class diagram for data connector hierarchy

Class DataServiceImpl contain two methods for loading the business data and also one method for generating update script from changes made in frontend. Changes are passed as list of objects with RdfChange type. After running this script on a SPARQL endpoint, changes made in frontend will persist permanently. And the last interface DataController also serves just for calling proper method in service layer.

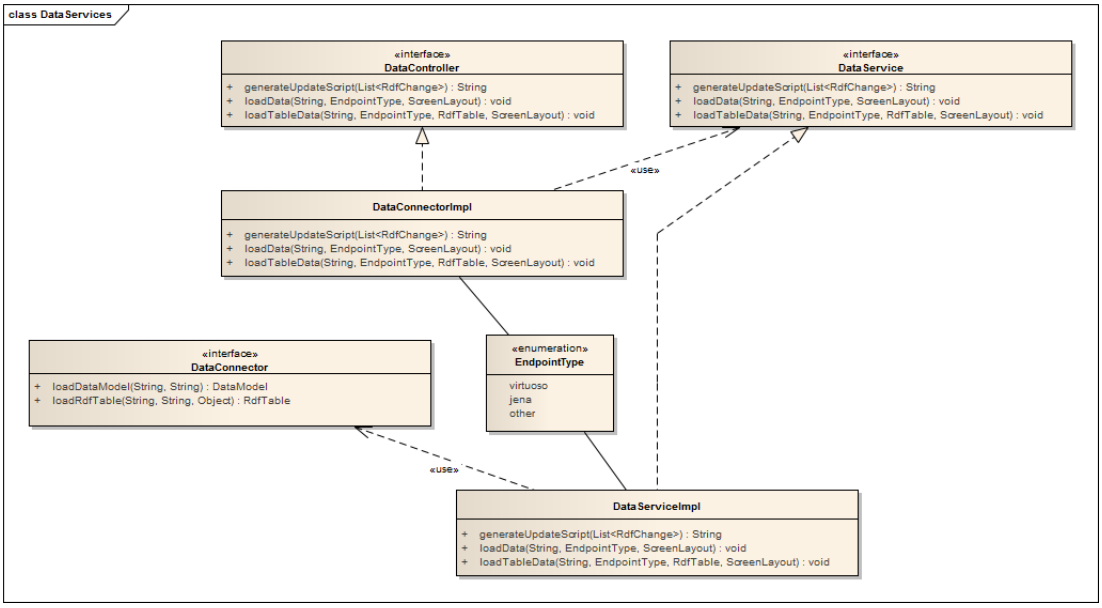

Figure 3.14: Class diagram for data service hierarchy

3.15 Used libraries and frameworks

There are several libraries that are used in the application. In following list there are all entities used in the application.

Name	Licence
Spring	Pivotal Software
Apache Jena	The Apache Software Foundation
Bootstrap	Twitter, Inc.
AngularJS	Google Inc.
jQuery	The jQuery Foundation

4. Testing

4.1 Unit testing

Every suitable class in the application should have its unit test. There are two types of objects - a data object and interface objects. As data objects serves only as a container, it doesn't make sense to write unit tests for these objects. On the other site, interface objects (with that we mean interfaces and classes implementing them) is more suitable for unit testing. They should be written in a way to check following things:

- correct methods are called between the tiers
- more complex business logic is done correctly
- incorrect values don't harm the application flow
- catching of exceptions

The better way to describe unit tests is within the source code. All unit test should be documented within the code, it should contain information about what the test is intended to. Summary of all existing unit test classes including individual methods in tests describes the following figure.

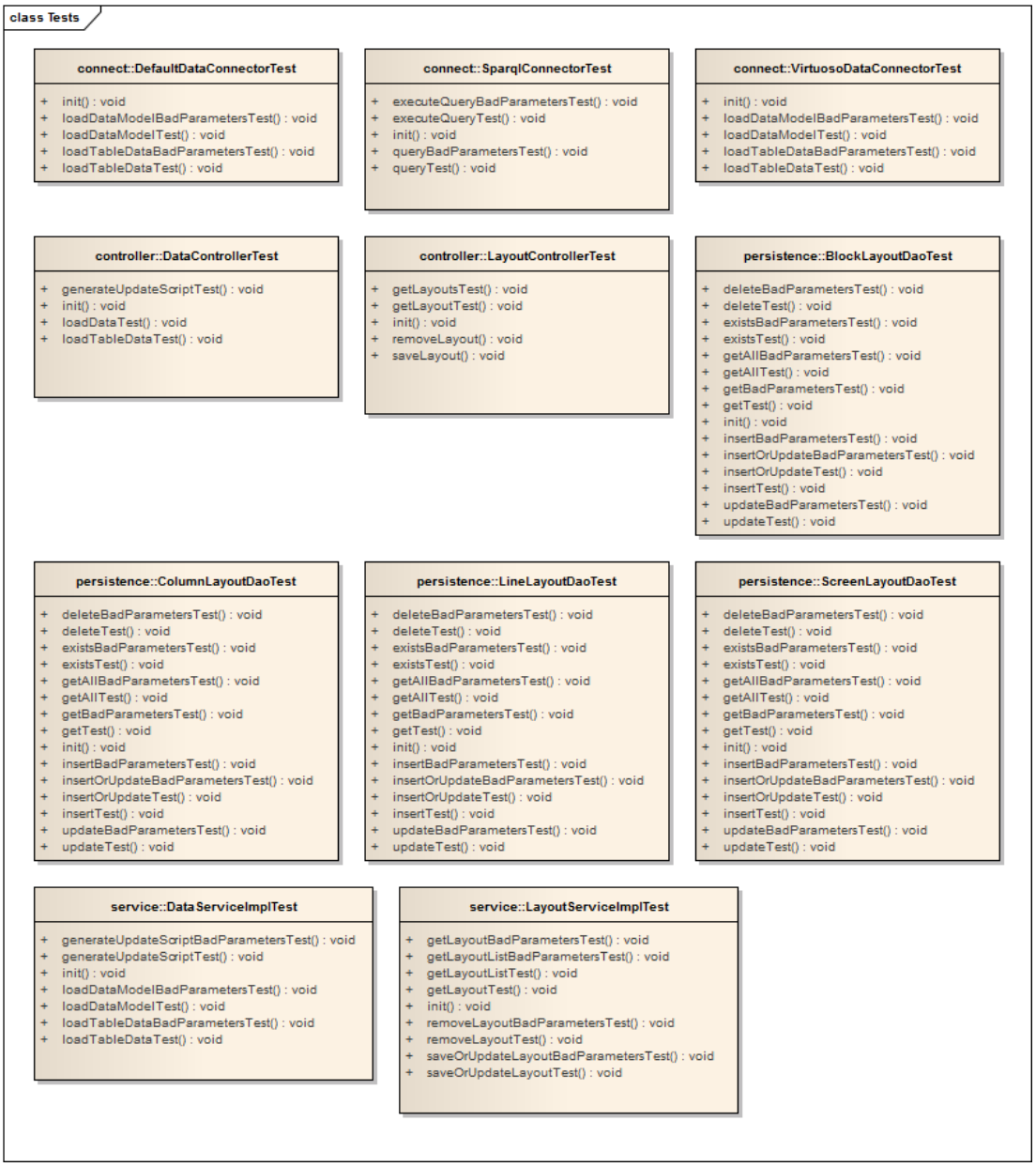

Figure 4.1: Unit tests class diagram

4.2 Acceptance criteria

Acceptance criteria will be based on use cases but don't fully match them. If all criteria are satisfied, we can say, that the application works correctly and satisfies requirements. All designed features have to be available as well. Follows the list of acceptance criteria:

Name	Create new layout definition
Criteria	1. The layout subpage contain a create button. 2. We can create an URI and a name for the layout and they're mandatory. 3. We can save a layout via a button. 4. After saving, the layout displays in the layout list.
Result description	The layout subpage contains the create button. After clicking an editing area appears. Layout name field is mandatory to fill in and layout can't be saved without entering a name. After saving a layout name successfuly, it appears in a list of layouts.
Result	Success.

Name	Use previously saved layout definition
Criteria	1. There is a list of saved layouts. 2. We can click a layout to show its detail. 3. Selected layout can be modified and saved. 4. All changes have to be visible after saving.
Result description	After entering the layouts subpage a loading icon is shown. After loading layout list from backend, the icon is hidden and a list of layouts is shown. After clicking a layout, an editing area appears and name of the selected layout and its content is shown. After modifying the name, or modifying the namespaces that can be accessed by click a special button, or modifying the layout settings as well as any object in editing area and saving changes, whole progress is saved. After saving a user goes back to layout list.
Result	Success.

Name	Create a table
Criteria	1. On a screen there is a create table button. 2. After clicking the button a new table appears. 3. Different kinds of properties can be set. 4. After confirming, all changes will shows on the screen. 5. After double clicking a table the modal window appears again and properties can be edited.
Result description	Clicking the button causes creating a rectangle in editing area representing a table. Some default properties are set. Clicking the rectangle opens a modal window where all object's properties can be changed. Behavior is dynamic and in realtime and as a user changes the properties in modal window, appearance of the rectangle changes in time.
Result	Success.

Name	Set position and size of the table
Criteria	1. The table modal windows contain fields for size and position definition. 2. After changing the position and size of the table change.
Result description	Size and position of a table represented by a rectangle can be changed in a modal window by entering a numeric values for particular properties. rectangle position can be also change by dragging the rectangle in the editing area. Editor doesn't allow to move a rectangle outside the editing area and watch its boundaries.
Result	Success.

Name	Create a line
Criteria	1. On a screen there is a create line button. 2. After clicking the button a new line appears and a modal window appears. 3. Source and target tables and property can be set. 4. We can't add two same lines - with same source, target and property.
Result description	Click the create line button causes a modal window to show. In that window there are list of existing tables that can be connected. Source, target and property fields are mandatory and have to be filled in order to create a connection. If there are less than two tables the connection cannot be created. The editor also watches the data correctness and connection from on table to the same table cannot be created as well as an existing connection but connection in the opposite direction is possible, which is correct. Connection between same sources and targets are possible, if the property connecting them is different. In modal window there are more properties to set.
Result	Success.

Name	Modifying a line
Criteria	1. After clicking a line the modal window appears again. 2. Properties can be edited and saved. 3. If source or target table changes, it have to be visible on a screen.
Result description	Clicking an existing line opens a modal window. Saved properties are filled in the window. In case of changing source and target tables, a line is redrawn. Other properties can be changed as well. Visual appearance of a line remains same - it's always black and red after mouse over because of keeping objects in editing are simpler, but properties are saved.
Result	Success.

Name	Set table properties
Criteria	1. The table modal window contain fields for columns definition. 2. We can add, edit or remove a column.
Result description	A modal windowing opened after clicking a rectangle contains a list of defined columns and there is a possibility to add a new one. There are three different types of columns that can be entered (URI, label and other custom property). Each column has its caption, which can be its URI or custom text. Column that is added to list can be modified or deleted.
Result	Success.

Name	Sort table rows
Criteria	1. After clicking a table column title, all table lines are sorted by that column. 2. Numeric values are sorted numerically and text values are sorted by text value. 3. After clicking a column again, the sort direction changes.
Result description	Clicking a table column caption causes reordering of the table. Columns are ordered relatively to their context alphabetically or numerically. A table rows can be sorted only by one column at once. After first clicking rows are sorted in ascending order and after second clicking they are sorted in descending order.
Result	Success.

Name	Filter table rows
Criteria	1. After entering a text in a column header only rows which particular column contain entered text appears. 2. This have to work for every column in a table. 3. If filter is empty, all values are visible.
Result description	Tables content is empty at first. To load some data a user have to enter some search text. After entering at least two characters, new data are loaded. New data are loaded after each changing of a filter field. The filter fields work independently for each column of a table. As a user removes a filter field content, rows of a table are appearing as a filter is less restrictive.
Result	Success.

Name	Show concrete class instances
Criteria	1. After selecting an instance by clicking it, in all adjacent tables there will appear only those instances which are connected to the selected one. 2. After clicking an instance again, the selection is cancelled and all entries will appear.
Result description	Clicking a concrete instance leads to filtering rows in all connected tables. It works for both directions - table is a source of some property or is a target of some property. Combination of selected instances from multiple tables has an effect and can be combined to obtain an optimal result. Clicking an instance also starts loading of new date in all connected tables. After deselecting an instance by clicking it again, the hidden rows in connected tables starts to appear.
Result	Success.

Name	Universality of sorting, filtering and showing concrete instances
Criteria	1. Rows affected by filter can be selected as instance. 2. Rows affected by selecting instance in source table can be filtered. 3. Rows affected by filter or by instance selection can be sorted by clicking a column title.
Result description	Different combination of column filters and selecting a table instance works. A user can combine both types of filter to achieve demanded result.
Result	Success.

Name	Reloading table content after filter changed
Criteria	1. Newly loaded rows are added to a table and are accessible after filter is canceled.
Result description	After changing any type or filer - both entering a column filter or selecting an instance causes loading of new data in attached tables. Based on layout settings the data is loaded only in directly attached tables or loading data in whole data structure.
Result	Success.

Name	Reloading tables content by selecting an instance
Criteria	1. In all adjacent tables a content is reloaded and newly loaded lines are added to tables.
Result description	After changing any type or filer - both entering a column filter or selecting an instance causes loading of new data in attached tables. Based on layout settings the data is loaded only in directly attached tables or loading data in whole data structure.
Result	Success.

Name	Marking of errors
Criteria	1. If there are any data errors a mark is displayed.
Result description	
Result	Success.

Name	Allow or disallow marked objects
Criteria	1. After confirming an error, the value remains and the error disappears. 2. After declining the error, the value and the error disappears.
Result description	
Result	Success.

Name	Modify literal value
Criteria	1. After right clicking a table cell, the context menu appears. 2. After choosing to remove cell, the value is deleted. 3. After choosing to edit cell, the new value will be shown on a screen.
Result description	Right clicking a table cell opens context menu containing commands for removing and editing table cell. Removing a cell content removes cell content and editing a cell causes change of a cell literal value. Both actions are bounded to cell that was right clicked.
Result	Success.

Name	Modify instances
Criteria	1. Choosing to remove the instance will causes the row to disappear. 2. Choosing to add an instance causes a modal window to appears. Values for all columns can be set and after confirming values a new row appears in the table. 3. Choosing to add an object property causes a modal window to appear. A reference to an instance in adjacent table can be set. If changes are confirmed and after selected the changed instance, in adjacent table a new entry is visible. 4. Choosing to remove an object property causes a modal window to appear. We can remove a property referenced in an adjacent table. After confirming the changes and selecting the changed instance, in adjacent table there is no longer removed instance reference visible.
Result description	Right clicking a table cell opens context menu containing commands for operating the instance. Removing the row from table causes that row to disappear. Removed line can be loaded again by a filter operation. Adding a new row opens a modal window where all row properties can be entered. The new line appears in a table immediately but it can be hidden by some filter right after adding. Entering row's URI is mandatory because it serves as an identifier. Adding an object property opens a modal window where a new property can be defined. There is a list of all defined connection leading from the table and after selecting the demanded one, a list of possible objects will appear. In that way a created object property is correctly defined. Removing an object property opens a modal window where some of properties can be removed. It is possible for many properties at once. Changes done for table rows are immediately propagated to data model and can be easily checked.
Result	Success.

Name	Save or cancel changes
Criteria	1. After clicking a save button an update script will appear and it corresponds to performed changes. 2. Update script can be downloaded as a file. 3. Committing the changes will save the progress in frontend and clears changes for next update script. 4. After clicking a cancel button all changes are cancelled and origin values will appear.
Result description	All changes can be discarded by clicking discard button. Current changes reverts and original data model shows. This includes data loaded by filters operations, only modified data are reverted. All performed changes can be shown as RDF script. That script can be downloaded as a file. Saving the changes stages performed changes and marks them as original data model. Next changes will be described from that point.
Result	Success.

5. User documentation

The application starts at homepage. There are two links for Layouts and Workbench. Layouts page contains list of saved layouts and in there a new layout can be created. Workbench page contains a working area where data are visualized into a static layout. Both pages can be accessed from menu from any page of the application.

5.1 Managing layouts

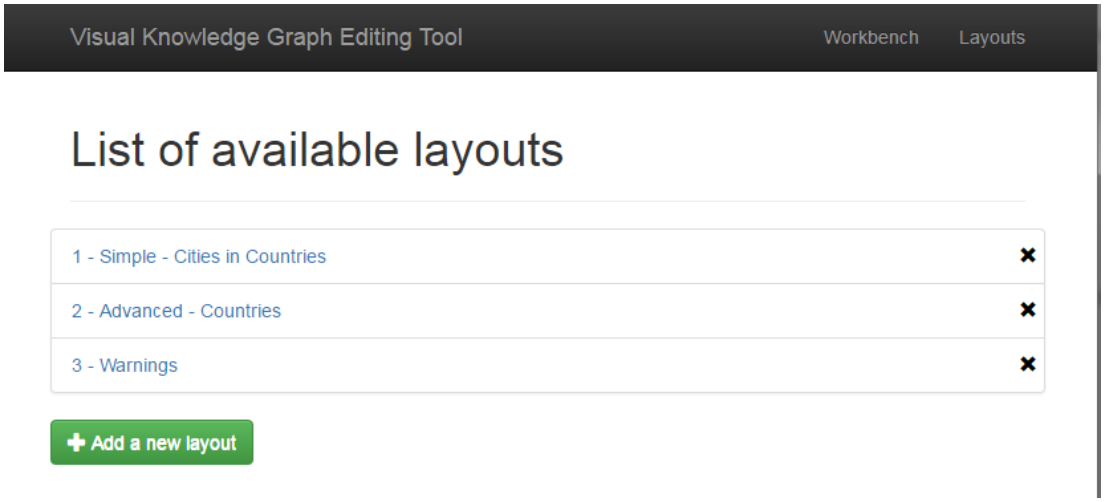

Figure 5.1: Layouts list page

In Layouts page we can do three actions - create a new layout, remove a layout and edit an existing layout. For creating a new layout, we click the "Add a new layout" button in a bottom part of the screen. For removing a layout, we click the "X" sign in the right part of the screen. For editing an existing layout we click a layout name.

5.2 Creating/editing a layout

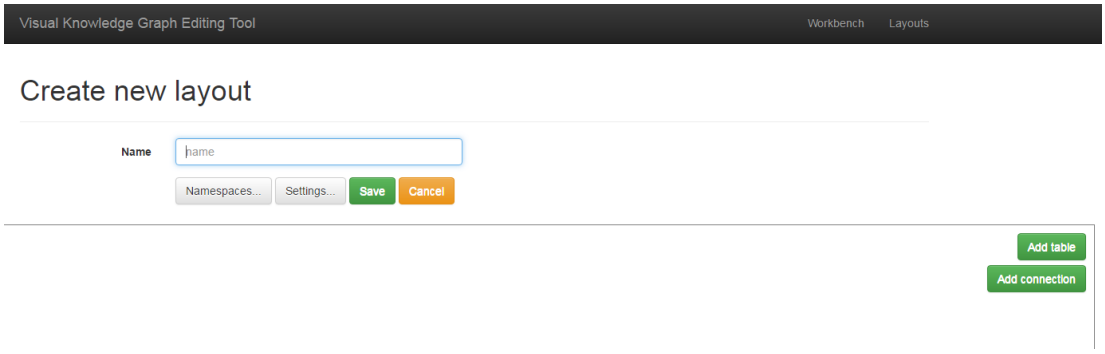

Figure 5.2: Create or edit a new layout

To create a new layout we can follow these steps. Working area for creating and editing a layout are identical and provides same operations.

1. Enter a layout name. This field is mandatory.

2. For managing a namespaces used in layout we click the "Namespaces..." button. Defining a namespace is useful for creating an abbreviation or a prefix for it and later in any properties we can use the abbreviation instead of full namespace. We can add a new namespace by entering it abbreviation

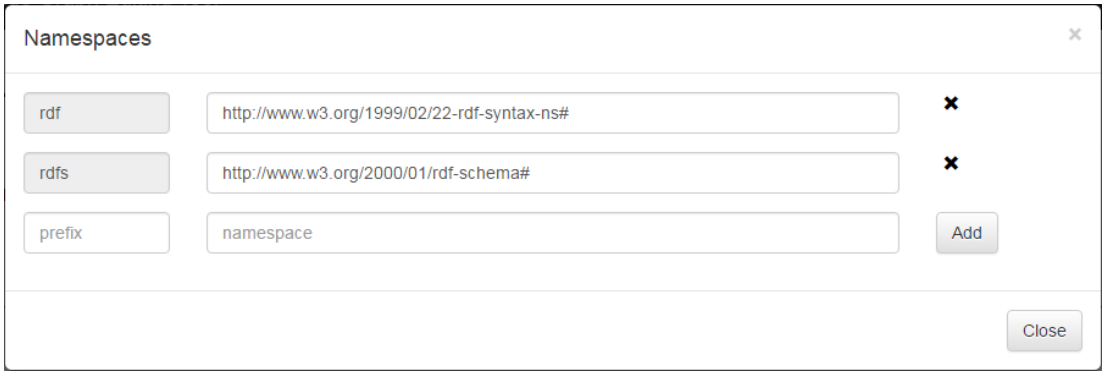

Figure 5.3: Namespaces management

   in the left field and its full identifier in the right field. After clicking the "Add" button a namespace will show in the list. We can remove a namespace by clicking the "X" symbol in the right part of the modal window. An abbreviation is fixed, but we can modify a namespace identifier.

3. To manage other properties, we click the "Settings..." button. Now it contains only one setting called "Filter propagation". It defines, how many tables will be affected by changing a filter by selecting an instance. "Neighbors" defines that only tables directly connected with the table with selected instance will be loaded with new data. The option "All" defines that all connected tables (even indirectly) will be reloaded.

4. Click "Cancel" button for discarding any changes.

5. Click "Save" button to save the layout.

6. If save was successful, the new layout is now available in layout list.

5.3 Creating/modifying a table

Clicking the "Add table" button will create a rectangle which represents a table. Newly created rectangle is a valid definition, but contains only default predefined data. We can drag and drop a rectangle to change its position on a screen. Position of a rectangle on a screen describes a position of a table in the working area in Workbench page. To edit its properties we double click the rectangle and a modal window will show. Modal window contains following properties to set.

For rdf:type This is a class of a table. We can enter whole identifier (for example "http://www.example.com/someType") or we can use a namespace prefix (for example "ex:someType") defined earlier.

Table label This property describes what should be used as table caption. Option "URI" simply shows value from the field above. Option "Constant" shows custom text defined in the field below "Label source". Option "Label" shows a label of the class of selected language. Option "Property" shows any other demanded property. In this case, in field "Label source" there will be a name of the property. And as well we can define a language for that property.

Columns There are three types of columns. URI, rdfs:label and custom. Column of type "URI" show instance's URI as is. Column of type "rdfs:label" shows value defined as "rdfs:label" for the instance. The last type shows any defined property. To define property name we can use full identifier or we can use a namespace prefix as well. Caption type describes what should be shown as column caption. Option "URI" will show a value as defined in the field on the left. Option "Constant" show custom text in the field of the right. If we choose a column of type "rdfs:label" we can define what language the label should be. In case of multiple value for a property, we can define what value to show by selecting an aggregate function option.

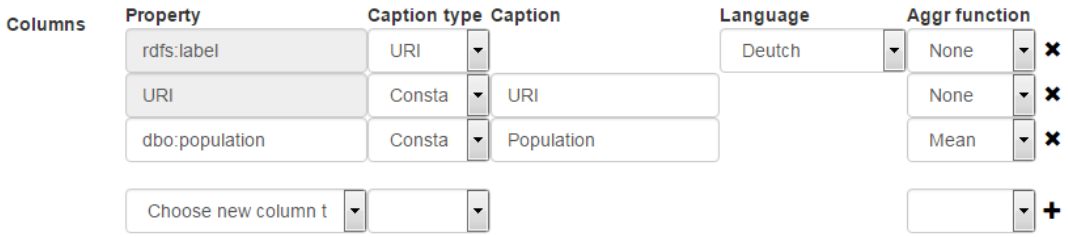

Figure 5.4: Columns in a table

Other properties Other properties describes visual appearance of a table - its position, sizes, border, color and font.

Remove To remove a rectangle we click the "X" symbol in the right top corner of a rectangle. All lines leading from or to a the rectangle will be removed as well.

5.4 Creating/modify a connection

Two create a connection between tables there must be at least two rectangles in the area or the connection will be unable to create.

1. To create a new connection we click the "Add connection" button. A modal window will appear.

2. Select source table in field "From type". In a RDF triple it describes a type of the subject.

3. Select destination table in field "To type". In a RDF triple it describes a type of the object.

4. Enter a property in the field "Property" connecting these two tables. We can enter whole identifier or we can use a namespace prefix defined earlier. In a RDF triple it describes a predicate.

5. Field "Label type" describes what to show as a title of a line in the working area. Option "URI" shows a value in "Property" field. Option "Constant" shows a custom text defined in the field below.

6. We can set visual appearance of the line connection tables, such as font, color, line type or thickness.

5.5 Start to work

To start the work we enter the Workbench page, even from the menu or from the home page. Now we have to define the data source and a layout defining demanded structure.

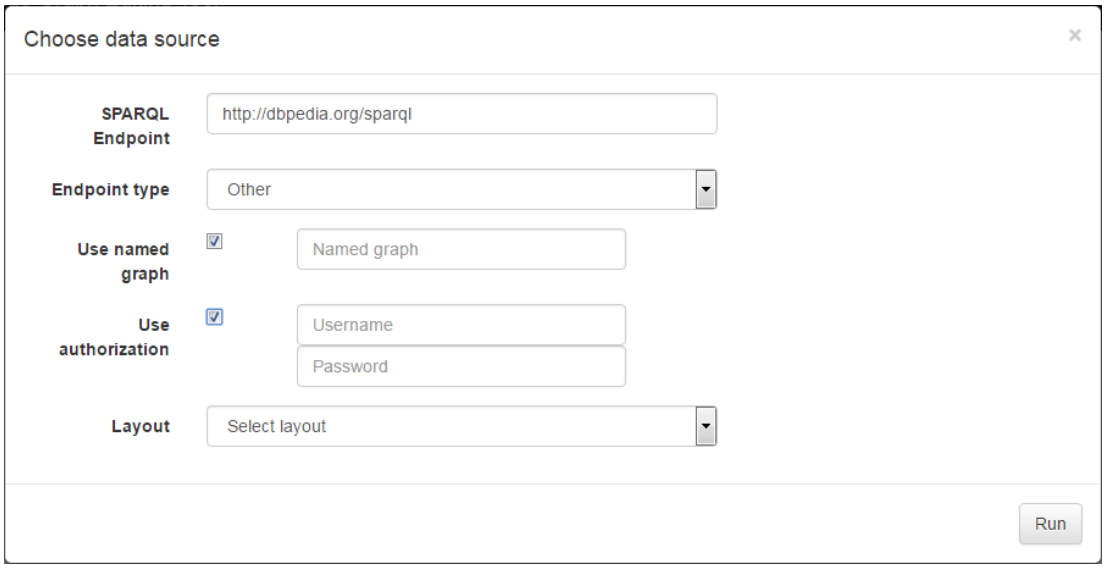

Figure 5.5: Data source

SPARQL Endpoint We enter a URL of the SPARQL Endpoint we want to use as a data source.

Endpoint type If we now a type of the SPARQL Endpoint, we can define it in this field. Special endpoint type contains functions specific for that type of endpoint and in that way can be faster. If we are not sure about endpoint type, we can just choose "Other" and the application will use common functions.

Named graph The field named graph is optional and if we want to use named graph, we check the checkbox and then enter a name of the named graph.

Authorization If the SPARQL Endpoint requires an authorization, we can check the checkbox and enter user name and password for the endpoint.

Layout Field "Layout" contains all stored layouts. From this list we choose a layout describing the data structure.

Run If all fields are set, we can click the "Run" button. After that a loading icon will show and the application will assemble a working area based on a layout definition and data from endpoint.

5.6 Working with tables

At the beginning tables content is empty. To load first data, we enter some text in "search..." fields within table columns headers. We have to type at least two characters to start the loading. It will search for instance by fulltext filter in relevant properties. The search is case sensitive.

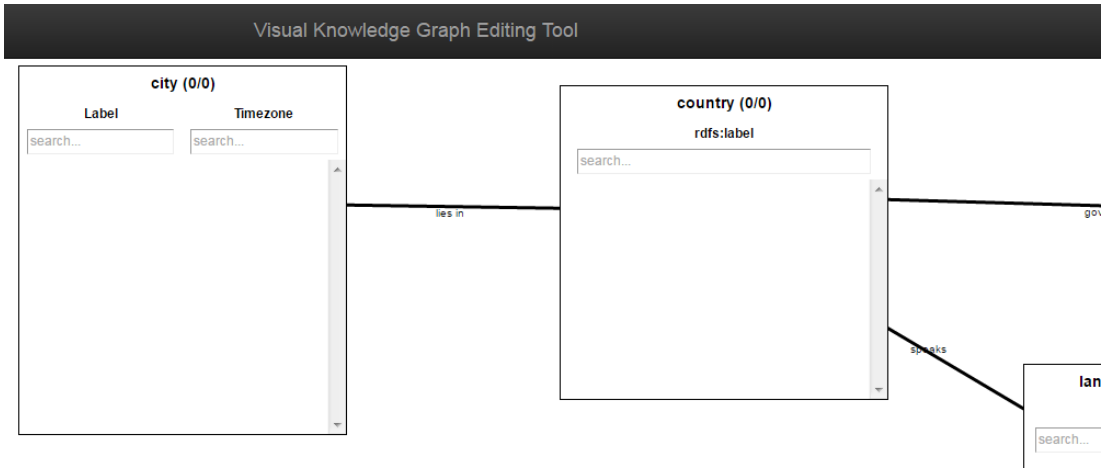

Figure 5.6: Working area

1. We can sort table rows by clicking the column header. It will sort rows by clicked column in ascending order. Second clicking on the column header will sort rows in descending order. If a content of cells is numeric, it will sort rows numerically. In the other case, it will sort them alphabetically.

2. Left clicking a table row will cause filtering by instance and loading relevant data. All adjacent tables will be reloaded and be filtered to show only relevant rows.

5.7 Edit data

To edit data, we right click a table cell after that a context menu will show.

1. Choosing the "Clear cell" action set the cell content empty.

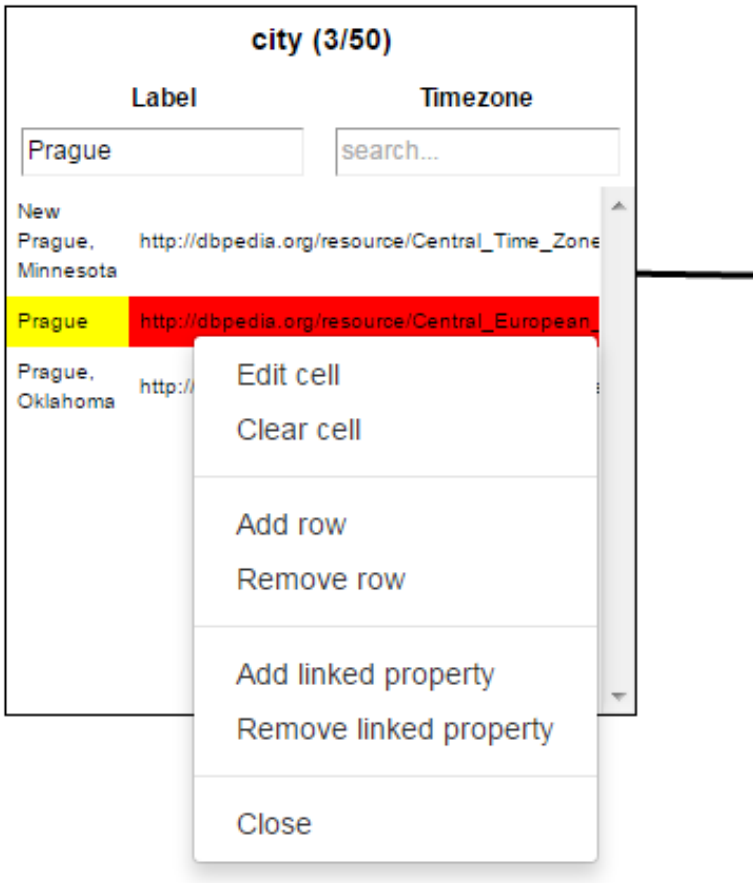

Figure 5.7: Context menu

2. Choosing the "Edit cell" action opens a model window where a new value can be set. We can see the old value in the upper field and we can set a new value in the bottom field. Change can be reverted by pressing the "Cancel" button or can be confirmed by pressing the "Save" button.

3. Choosing the "Add row" action will open a model window where values for a new row can be set. In the modal window there are all properties contained in the table. The "URI" field is mandatory because it is a new row's identifier. Other fields are voluntary. We can cancel change by clicking the "Cancel" button or confirm the changes by clicking the "Save" button. After saving the changes a new row is inserted to line immediately and all

Figure 5.8: Edit cell modal window

Figure 5.9: Add a new row

active filters are activated to him.

4. Choosing the "Remove row" action removes the selected row from the table immediately.

5. Choosing the "Add linked property" action opens a modal window where a new reference to an adjanced table can be set. At first, there is only

Figure 5.10: Add a new linked property

one field where we can choose which linked property leading from the table we want to set. After choosing a second field appear. In the second field we can choose one of loaded instance from target table. Only a loaded instance can be added. If there is no demanded instance we have to load it by filters operations. We can cancel change by clicking the "Cancel" button or confirm the changes by clicking the "Save" button.

6. Choosing the "Remove linked property" action opens a modal window where existing linked properties can be removed. There is a list of all properties that leads from selected table to another. If there is no such a property the list will be empty. We check all properties we want to remove and we can cancel change by clicking the "Cancel" button or confirm the changes by clicking the "Save" button.

7. Choosing the "Close" action will close the context menu.

5.8 Managing the changes

All changes we've done in work area can be discarded or saved.

Figure 5.11: Remove linked property

1. We click "Discard changes" button to revert all performed changes.
2. Clicking the "Manage changes" button opens a modal window. In text area

Figure 5.12: Manage changes

there is an update script which can be run in an SPARQL endpoint and performs same data modifications as we did in the working area. We can download script as a file by clicking the "Save..." button. By clicking the "Save changes" button all changes are saved to working data model. By clicking the button changes won't be persisted to endpoint serving as a data storage. It only saves changes done in the current data model.

Conclusion

The thesis was made in a classic waterfall methodic of software development. At first we summarized thesis requirements and analyzed them. Many consequences raised that had to be done in a development. The software was at first described and designed using UML language. A web application using web services was written and tests were made to assure that all requirements are fulfilled. Also a user tutorial were written to make a work more familiar to users.

Two main targets was to display data from SPARQL endpoint to a tabular structure and to fix possible errors via data modifications. The application is capable to load any type of RDF data and contains tool for defining layouts that describe a way to visualize them by user demands. The application can even handle larger amounts of data because they are loaded lazily in small bulks and are merged together later. Data modification can be done. A user can edit simple literal values or even references to other objects and are sufficient to perform any demanded changes.

Future work

User management

The application lacks the management of users. There are no ways to perform user authorization or authentication (that is only for accessing SPARQL endpoints but not the application itself). In future steps of development this can be done by several ways. First is to implement it by its own. More perspective way is to use third parties solution. Its advantage is that the authorization and authentication can be shared between other applications using the solution. Also, the application doesn't have to keep users data, just uses resources given the other solution.

Pre-filling classes and properties

While defining a layout user have to fill RDF classes and properties as simple text and has no control that they are correct. Next version of the application may support loading of all classes from some endpoint. After selecting a RDF class from available list, in a same way a list of properties for that class will load.

RDF script optimization

The RDF script that is a result of changes performed by a user is assembled by simple summarizing all changes in order without any modifications. It is clear that some optimization can be done. For example grouping many updates of same resource to one. Or in sequence of operation update - insert - delete over same resource first two of them don't have to be performed as at the end there will be removed. Optimizing the script is a difficult operation, because it have to ensure that the result gives exactly the same result as non-optimized script.

Persisting of changes

Now the only output that the application is able to produce from changes is the text script. A user have to process the script by himself. In a future version there can be tool which will persist the changes to some SPARQL endpoint or any other database.

Namespace checking

In layout editor we can define namespaces and they abbreviation as prefix. In other fields in editor there can be filled these prefixes to identify some property. The problem is that there is no check that used prefix is defined in namespace list. A future version may contain such a check. If a user enters a non-existing prefix a warning will be shown and a popup window may appear to enter the namespace definition immediately.

Bibliography

[1] Alessandro Bozzon, Marco Brambilla, Emanuele Della Valle, Piero Fraternali, Chiara Pasini. *A Conceptual Framework for Linked Data Exploration. Current Trends in Web Engineering Lecture Notes in Computer Science Volume 7059.* 2012. pp 109-118.

[2] *Linked Data.* [online]. [Accessed 26th July 2016]. Available from: http://linkeddata.org

[3] Christian Bizer, Tom Heath, Tim Berners-Lee. *Linked Data - The Story So Far. Int. J. Semantic Web Inf. Syst. 5(3).* [online]. 2009. pp 1-22. Available from: http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf

[4] Daniel Hienert, Benjamin Zapilko, Philipp Schaer, Brigitte Mathiak. *Vizgr: Linking Data in Visualizations. Web Information Systems and Technologies Lecture Notes in Business Information Processing Volume 101.* 2012. pp 177-191.

[5] *RDF Schema 1.1.*. [online]. Last revision 25th February 2014 [Accessed 26th July 2016]. Available from: https://www.w3.org/TR/rdf-schema/

[6] *SPARQL 1.1 Overview.* [online]. Last revision 21th May 2013 [Accessed 26th July 2016]. Available from: https://www.w3.org/TR/sparql11-overview/

[7] FIKES, Richard, Pat HAYES and Ian HORROCKS. *DAML Query Language (DQL).* [online]. Last revision April 2013 [Accessed 26th July 2016]. Available from: http://www.daml.org/2003/04/dql/dql

[8] *N3QL - RDF Data Query Language.* [online]. Version 1.66, 3rd July 2004 [Accessed 26th July 2016]. Available from: https://www.w3.org/DesignIssues/N3QL.html

[9] SEABORNE, Andy. *RDQL - A Query Language for RDF.* [online]. Last revision 9th September 2004 [Accessed 26th July 2016]. Available from: https://www.w3.org/Submission/RDQL/

[10] GAMMA, Erich. *Design patterns: elements of reusable object-oriented software.* Boston: Addison-Wesley, c1995. Addison-Wesley professional computing series. ISBN 0-201-63361-2.

[11] Whetzel PL, Noy NF, Shah NH, Alexander PR, Nyulas C, Tudorache T, Musen MA. *BioPortal: enhanced functionality via new Web services from the National Center for Biomedical Ontology to access and use ontologies in software applications.* Nucleic Acids Res. 2011 Jul;39(Web Server issue):W541-5. Epub 2011 Jun 14.

[12] *OntoStudio.* Semafora Systems 2012. [Accessed 26th July 2016]. Available from: http://www.semafora-systems.com/en/products/ontostudio/.

[13] *Protégé*. [online]. Stanford Center for Biomedical Informatics Research (BMIR) at the Stanford University School of Medicine. 2016 [Accessed 26th July 2016]. Available from: http://protege.stanford.edu.

[14] *OpenData.cz*. [online]. 2016 [Accessed 26th July 2016]. Available from: http://www.opendata.cz.

List of Figures

List of Abbreviations

DAO	Database access object. A design pattern providing an abstract interface to a persistence mechanism.
DQL	Data query language.
JSON-LD	JSON for Linked Data. A lightweight Linked Data format.
MFF	Faculty of Mathematics and Physics.
MVC	Model-view-controller. A software architectural pattern.
N3QL	A RDF query language derived from Notation 3 Logic and RDQL.
RDFS	Resource definition framework schema.
RDF	Resource definition framework.
RDQL	A Query Language for RDF
SPARQL	A recursive abbreviation for SPARQL Protocol and RDF Query Language. A query language for RDF.
TTL	Terse RDF Triple Language. A RDF format.
UC	Use case.
UK	Charles University in Prague.
UML	Unified modeling language.
URI	Uniform resource identifier. A text string identifying a resource which consists of an URL and a resource name.
URL	Uniform resource locator. A web address identifying a web resource.
W3C	World Wide Web Consorcium.
XML	Extensible markup language.